Introduction to MATLAB / SIMULINK

C. James Taylor
c.taylor@lancaster.ac.uk

Engineering Department
Faculty of Science and Technology
Lancaster University

MATLAB™ is an interactive programming language that can be used in many ways, including data analysis and visualisation, simulation and engineering problem solving. It may be used as an interactive tool or as a high level programming language. It provides an effective environment for both the beginner and for the professional engineer and scientist. SIMULINK™ is an extension to MATLAB that provides an iconographic programming environment for the solution of differential equations and other dynamic systems.

The package is widely used in academia and industry. It is particularly well known in the following industries: aerospace and defence; automotive; biotech, pharmaceutical; medical; and communications. Specialist toolboxes are available for a diverse range of other applications, including statistical analysis, financial modelling, image processing and so on. Furthermore, *real time* toolboxes allow for on-line interaction with engineering systems, ideal for data logging and control.

At Lancaster University, MATLAB is used for research and teaching purposes in a number of disciplines, including Engineering, Communications, Maths & Stats and Environmental Science. In Engineering, students use MATLAB to help with their coursework, 3rd year individual project and MEng team project, as well as in their later career.

References

These notes are based on the Laboratory Handouts for the following Engineering Department taught modules:

- ENGR.202 Instrumentation & Control
- ENGR.500 MSc Start–Up Week

Some sections align with the ENGR.202 syllabus (particularly the use of a generalised second order differential equation in Part 3) but most examples are designed to be self-explanatory.

The notes are also based on the laboratories for ENGR.263 System Simulation (now laid down), developed by Prof. A. Bradshaw and updated by the present author.

Some of the examples are based on code from the following recommended textbook:

- Essentials of Matlab Programming (2009) S. J. Chapman, *Cengage Learning*, International Student Edition, 2nd Edition.

MATLAB/SIMULINK™ is developed and distributed by The Mathworks Inc.

For more information visit their web site at: http://www.mathworks.com/

These notes are based on MATLAB 7.0 (R14), running from a Windows XP based PC.

Differences may emerge in other versions of the software, but trial and error experimentation should usually solve any problems.

27th March 2010

Part 1 – Introduction to MATLAB

1.1 Learning Objectives

This laboratory (Part 1) provides a basic introduction to MATLAB. By the end of the session, you should be able to:

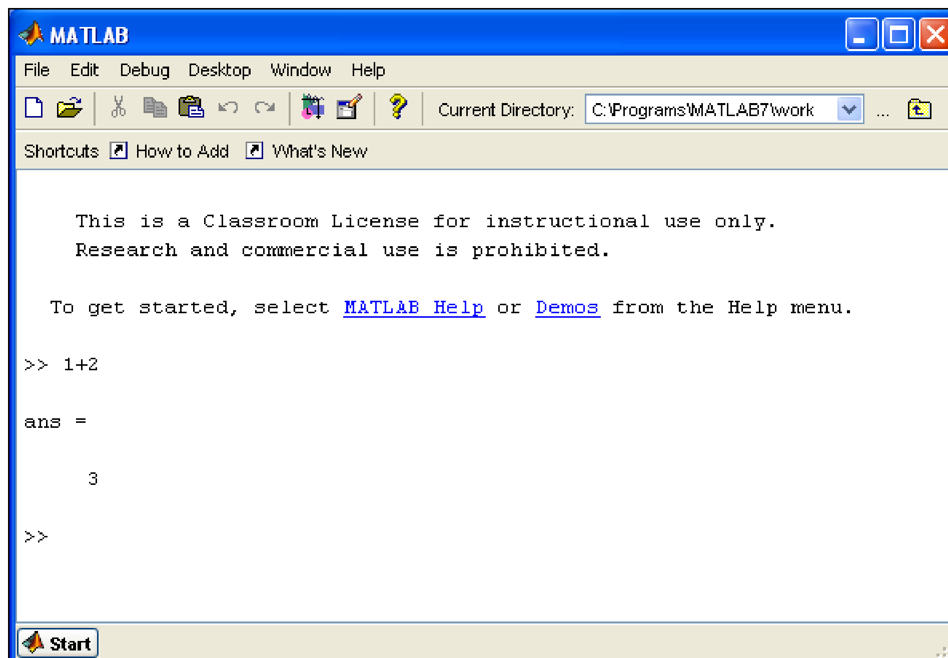
- use MATLAB in 'calculator mode';
- manipulate variables and import data files;
- plot, annotate and copy graphs to a word processor;
- write simple programs (scripts) using loops and conditional statements;
- use in-built MATLAB functions, e.g. `cos`, `sin` and `plot`.

Before leaving the class, check back on these objectives – these skills will be needed later on!

1.2 MATLAB Command Line

Although its original inspiration was to provide easy access to matrix operations, MATLAB ('Matrix Laboratory') can also be conveniently used for elementary calculations such as those available on an electronic calculator, as shown below. To begin a session on a PC, click on the MATLAB desktop icon or use the Windows Start menu. The "MATLAB Command Window" will eventually appear, where you can type instructions. Depending on the last user of your PC, the interface may appear differently. To avoid later confusion in these notes, you should follow the steps below before continuing:

- Select **Desktop** from the MATLAB menu
- Select **Desktop Layout → Command Window Only**



The MATLAB window should now look something like the picture above. Type `1 + 2` and press return. The result is assigned to the generic variable `ans` and is printed on the screen. Not too difficult so far! You can quit MATLAB at any time by selecting the File Menu item Exit or by typing Quit in the command window.

1.2.1 Expressions

The usual arithmetical operators,

+	addition
-	subtraction
*	multiplication
/	division
^	power

can be used in expressions. The rules of precedence are ^ first, then / , * , - and finally +. Expressions in brackets are evaluated first. Where two operations rank equally in the order of preference, then the calculation is performed from left to right. Blank spaces around the = , + and - signs are optional. Spaces are used in these notes to improve readability, but you do *not* need to type these spaces out when you try the examples for yourself.

Arithmetic expressions allow MATLAB to be used in 'calculator mode' as shown below, always remembering to press return after typing a command:

```
» 100/5    < return >
ans =
     20
» 100/5/2
ans =
     10
» 100 + 5/2
ans =
     102.5
» (100 + 5)/2
ans =
     52.5
» 100 + 5^2/2
ans =
     112.5
» 100 + 5^(2/2)
ans =
     105
```

In all these examples, the result is assigned to the generic variable `ans`. Variables can be used in expressions if they have previously been assigned a numerical value, as in the following example:

```
» a = 10
» b = 100
» a*b
ans =
     1000
```

Variables can be reassigned:

```
» a = 10
» b = 100
» a = a*b
a = 1000
```

Use the `who` command to see a list of defined variables in alphabetical order. If you have been typing the commands above, you will have three variables in memory so far: `a`, `ans` and `b`. Type the name of one of these variables and press return to see its current value:

```
» who
Your variables are:
  a  ans  b
» a
a =
     1000
» b
b =
     100
```

Exercise 1

Use MATLAB to evaluate the following:

$$\textbf{(i)}\quad 100+5^3, \qquad \textbf{(ii)}\quad \frac{100+5}{2+10}, \qquad \textbf{(iii)}\quad 3\times 100+\frac{97\times 5}{2+20}$$

1.2.2 Editing previous commands

MATLAB responds to invalid expressions with a statement indicating what is wrong, as in the example below. Here, we would like to sum two variables and then divide the total by 2, but what happens if we miss out the closing right bracket?

```
» b = 100
» c = 5
» (b + c/2

??? (b + c/2
A closing right parenthesis is missing.
Check for a missing ")" or a missing operator.
```

At any time, you can use the cursor keys to:

- edit text on the command line (using the **left / right** arrow keys);
- scroll back to find previously entered commands (using the **up / down** arrow keys);

Exercise 2

Use the arrow keys to edit the previous command and hence find `(b+c)/2`

Check that the answer makes sense.

1.2.3 Statements and variables

Statements have the generic form: `variable = expression`

The equals symbol implies the assignment of the expression to the variable. Several scalar examples have already been given above, while a typical vector statement is:

```
» x = [1 3]
x =     1     3
```

Here the numbers 1 and 3 are assigned to an array or vector (i.e. a list of numbers) with the variable name `x`. The statement is executed immediately after the enter key is pressed. The array `x` is automatically displayed after the statement is executed. If the statement is followed by a semi-colon then the array is not displayed. Thus, now typing the statement:

```
» x = [1 4];
```

would not result in any output on screen but *the assignment will still have been carried out!* You can confirm this for the above example by checking the current value of the variable:

```
» x
x =
     1     4
```

Although not often required in these laboratory notes, the semi-colon is useful when analysing large arrays, since it avoids having to wait for several pages of data to scroll down the screen. It is also useful later on when writing your own functions, since you can avoid displaying unnecessary intermediate results to the screen.

Array elements can be any valid MATLAB expression. For example:

```
» x = [-1.3  3^2  (1+2+3)*4/5]
x =
   -1.3000    9.0000    4.8000
```

Individual array elements can be referenced with indices inside brackets. Thus, continuing the previous example:

```
» a = x(2)
a =
    9
» x(1)
ans =
      -1.3
» -2*x(1)
ans =
      2.6

» x(5) = -2*x(1)
x =
   -1.3000    9.0000    4.8000         0    2.6000
```

In the last case, notice that the size of the array has been automatically increased to accommodate the new element. Any undefined intervening elements, in this case `x(4)`, are set to zero.

1.2.4 Elementary functions

All the common trigonometric and elementary mathematical functions are available for use in expressions. An incomplete list includes:

`sin(X)`	sine of the elements of `X`
`cos(X)`	cosine of the elements of `X`
`asin(X)`	arcsine of the elements of `X`
`acos(X)`	arccosine of the elements of `X`

`tan(X)`	tangent of the elements of `X`
`atan(X)`	arctangent of the elements of `X`
`abs(X)`	absolute value of the elements of `X`
`sqrt(X)`	square root of the elements of `X`
`imag(X)`	imaginary part of the elements of `X`
`real(X)`	real part of the elements of `X`
`log(X)`	natural logarithm of the elements of `X`
`log10(X)`	logarithm base 10 of the elements of `X`
`exp(X)`	exponential of the elements of `X`

These functions can be included in any expression. Note that the argument `X` of the function may be a scalar or an array. In the latter case, the result is an array with element-by -element correspondence to the elements of `X`, as can be seen in this example:

```
» sin([0 1])
ans =
     0    0.8415
```

The answer is in radians. Variable names begin with a letter that may be followed by any number of letters and numbers (including underscores), although MATLAB only remembers the first 31 characters. It is good practice to use meaningful names, but not to use names that take too long to type. Since MATLAB is case sensitive, the variables `A` and `a` are different. Note that all the predefined functions, such as those listed above have lowercase names.

```
» a = [0 1]
» A = sin(a)
A =
     0    0.8415

» B = cos(a)
B =
     1    0.5403
```

The function `clear` removes a variable from the workspace:

```
» clear A
» who
Your variables are:
  a  B  ans
```

MATLAB has several predefined variables, including:

pi	π
NaN	not - a - number
Inf	∞
i	$\sqrt{-1}$
j	$\sqrt{-1}$

Although it is not recommended, these predefined variables can be overwritten. In the latter case, they can be reset to their default values by using the `clear` function. For example:

```
» pi = 5
pi =
    5
```

Oops, probably a bad idea!

```
» clear pi
» pi
pi =
    3.1416
```

The special variable called `NaN` results from undefined operations. For example:

```
» 0/0
Warning: Divide by zero
ans =
     NaN
```

Variables are stored in the *workspace*. The `who` function introduced above gives a list of the variables in the workspace, while the `whos` function gives additional information such as the number of elements in an array and the amount of memory occupied. Typing `clear` by itself *removes all variables from the workspace*, while `clear name1 name2 ...` removes only the particular named variables in the list.

All computations in MATLAB are performed in *double precision*. However, the screen output can be displayed in several formats. The default contains four digits past the decimal point for non integers, as seen above. This can be changed using the format function as shown below:

```
» format long
» pi
ans =
   3.14159265358979
» format short e
» pi
ans =
  3.1416e+000
 » format long e
» pi
ans =
    3.141592653589793e+000
```

Finally, return to the standard format:

```
» format short
» pi
ans =
    3.1416
```

1.2.5 Array operations

Arrays with the same number of elements can be added and subtracted. This means adding and subtracting the corresponding elements in the arrays, as in the following example:

```
» x = [1 2 3]
» y = [4 5 6]
» z = x + y
z =
     5     7     9
```

This is called an *element-by-element* operation. Such operations are useful for setting up tables of values and for graph plotting. Attempting to perform element-by-element operations on arrays containing different numbers of elements will result in an error message.

Addition, subtraction, multiplication and division of an array by a scalar (an array with one element) is allowed and results in the operation being carried out on every element of the array. Thus, continuing the previous example:

```
» w = z - 1
w =
     4     6     8
```

Larger arrays can be set up by using the *colon* notation to generate an array containing the numbers from a starting value `xstart`, to a final value `xfinal`, with a specified increment `xinc`, by a statement of the form: `x = [xstart: xinc: xfinal]`. The following example generates a table of x against y where $y = x \sin(x)$.

```
» x = [0: 0.1: 0.5]
x =
         0    0.1000    0.2000    0.3000    0.4000    0.5000
» y = x.*sin(x)
y =
         0    0.0100    0.0397    0.0887    0.1558    0.2397
```

When the element-by-element operations involve multiplication, division and power, the operator is preceded by a dot, as shown below. *It is easy to forget this and encounter an error*!

```
» A=[1 2 3]
» B=[-6 7 8]
» A.*B
ans =
    -6    14    24
» A.^2
ans =
     1     4     9
```

The dot avoids ambiguities which would otherwise arise since, by default, MATLAB expects vector-matrix analysis: see below.

1.2.6 Vectors and Matrices

In vector-matrix terms, it is not possible to multiply two row vectors, hence simply typing `A*B` (without including the dot) results in an error:

```
» A=[1 2 3]
» B=[-6 7 8]
» A*B
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

However, a row vector can be multiplied by a column vector as follows. First of all, take the transpose of `B` using the inverted comma symbol, to create a column vector:

```
» C = B′
C =
    -6
     7
     8
```

In vector-matrix analysis, the order of the multiplication makes a difference to the solution:

```
» A * C
ans =
    32
» C * A
ans =
    -6   -12   -18
     7    14    21
     8    16    24
```

If you are not sure about the above results, then you might wish to read up about vectors and matrices. Matrices can be defined by hand using a semi-colon to indicate a new row:

```
» X=[1 2; 3 4; 5 6]
X =
     1     2
     3     4
     5     6
```

The transpose operator also works for matrices, as shown below:

```
» X′
ans =
     1     3     5
     2     4     6
```

Some matrices can be defined using built-in MATLAB functions, as in the following examples:

```
» ones(2, 3)
ans =
     1     1     1
     1     1     1
» zeros(2, 2)
ans =
     0     0
     0     0
» eye(3, 3)
ans =
     1     0     0
     0     1     0
     0     0     1
```

The last example is called the identity matrix.

Exercise 3

(i) Experiment with the functions `ones`, `zeros` and `eye` to learn how they work. Use these functions to create the following:

- 5 x 5 diagonal matrix with elements all equal to 8
- 6 x 6 matrix with all its elements equal to 3.5

(ii) Evaluate the following expressions, with: **A** = 100, **B** = 5, **C** = 2, **D** = 10.

$$\text{(i)}\ \frac{\mathbf{A}+\mathbf{B}+\mathbf{C}}{2\mathbf{D}},\qquad \text{(ii)}\ \mathbf{A}+\mathbf{B}^{\mathbf{C}},\qquad \text{(iii)}\ \frac{-1}{\mathbf{B}(\mathbf{A}-\mathbf{B})},\qquad \text{(iv)}\ \frac{\mathbf{AD}}{\mathbf{BC}}$$

(iii) Calculate the areas of circles of radius 1, 1.5, 2, 2.5, ..., 10m. Hint: to quickly solve all five cases at once, first define an array for the radius, e.g. `rad = [1:0.5:10]`.

(iv) Calculate the areas of the rectangles whose lengths and breadths are given in the following table. Again, it is good practice to use arrays – for more advanced problems, this would save the programmer a lot of time.

length	5	10	3	2
breadth	1	5	2	0.5

1.2.7 Graphics

Graphics play an important role in the design and analysis of engineering systems. The objective of this section is to introduce the most basic x-y plotting capability of MATLAB. A figure window is brought up automatically when the **`plot`** function is used. The user can switch from the figure window to the command window using the mouse. Multiple plots may be open at one time: use the **`figure`** command to open a new figure window.

Available plot functions include:

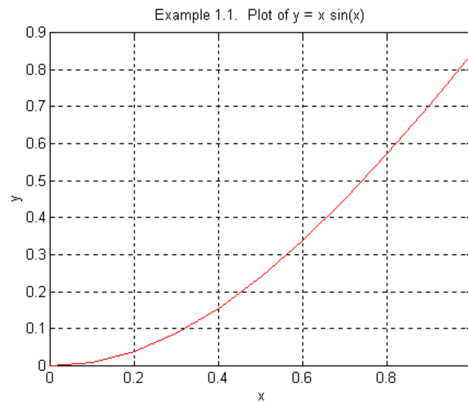
`plot(x,y)`	plots the array x versus the array y
`semilogx(x,y)`	plots the array x versus the vector y with $\log_{10}$ scale on the x-axis
`semilogy(x,y)`	plots the array x versus the vector y with $\log_{10}$ scale on the y-axis
`loglog(x,y)`	plots the array x versus the vector y with $\log_{10}$ scale on both axes

The axis scales and line types are automatically chosen. However, graphs may be customised using the following functions:

`title('text')`	puts 'text' at the top of the plot
`xlabel('text')`	labels the x-axis with 'text'
`ylabel('text')`	labels the y-axis with 'text'
`text(p,q,'text','sc')`	puts 'text' at (p, q) in screen co-ordinates
`subplot`	divides the graphics window
`grid on / grid off`	draws grid lines on the current plot (turns on or off)

Screen co-ordinates define the lower left corner as (0.0, 0.0) and the upper right as (1.0, 1.0). Plots may also be annotated by using the various menu options on the graph window. To illustrate some of these functions, consider a plot of $y = x\sin(x)$ versus x as shown below.

```
» x = [0: 0.1: 1.0]; y = x.*sin(x)
» plot(x, y)
» title('Example 1.1. Plot of y = x sin(x)')
» xlabel('x'); ylabel('y'); grid on
```

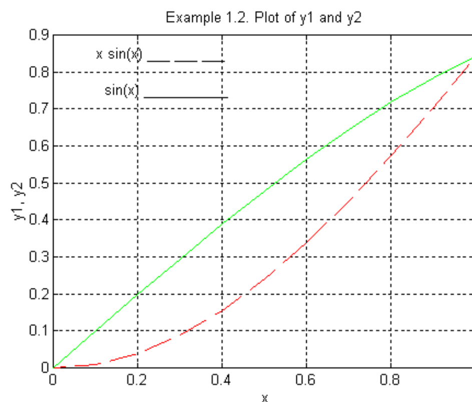


Plots may include more than one line and line types may be specified in the plot statement:

-	solid line	--	dashed line	:	dotted line
r	red line	b	blue line	k	black line

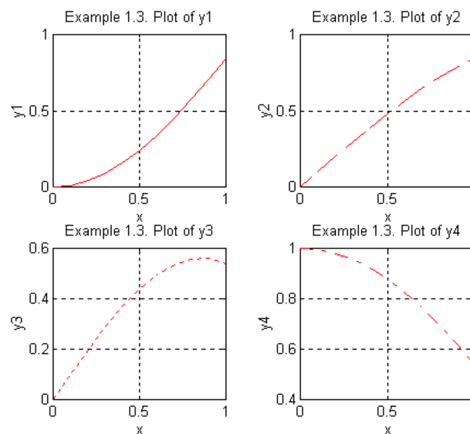
Type `help plot` to see a full list of all the colour and line type options. Normally, the `plot` command clears any previous lines in the same figure window. However, `hold on` freezes the figure and is useful for graphing multiple lines, as shown in the next example. Here, for brevity, some of the commands are written on the same line, separated by a semi-colon – you can either type the example this way, or write on separate lines as usual.

```
» x = [0: 0.1: 1.0]; y1 = x.*sin(x); y2 = sin(x);
» plot(x,y1,'--'); hold on; plot(x,y2,'-.')
» title('Example 1.2. Plot of y1 and y2')
» xlabel('x'); ylabel('y1, y2')
» text(0.1,0.85,'y1 = x sin(x) ---')
» text(0.1,0.75,'y2 = sin(x) -.-.-.')
```



The graph display can be divided into two, four or more smaller windows using the `subplot(m,n,p)` function. The effect is to divide the graph display into an *m* by *n* grid of smaller windows. This facility is illustrated in the next example. For brevity, the various annotations are omitted from the code below.

```
» x = [0: 0.1: 1.0];
» y1 = x.*sin(x); y2 = sin(x); y3 = x.*cos(x); y4 = cos(x);
» subplot(2,2,1); plot(x,y1,'-')
» subplot(2,2,2); plot(x,y2,'--')
» subplot(2,2,3); plot(x,y3,':')
» subplot(2,2,4); plot(x,y4,'-.')
```



This document was created using Microsoft Word. The MATLAB graphs where copied to the clipboard using the MATLAB Figure Window Menu → Edit → Copy Figure, before being pasted into the word processor.

To avoid ink wastage, it is best to set the background to white as follows:

- From the Figure Window Menu → Edit → Copy Options…
- Select Figure Copy Template → Copy Options → Force White Background.

Exercise 4

Plot the graph which relates temperature in degrees centigrade from –50 to 150 C to degrees Fahrenheit. Plot the graph of the inverse function which relates degrees Fahrenheit to degrees centigrade. The relationship between the two is:

$$Fehrenheit = 32 + \frac{9 \times Centigrade}{5}$$

1.3 MATLAB Scripts

So far, all interaction with MATLAB has been at the command prompt labelled ». At this prompt, a statement is entered and executed when the enter key is pressed. This is the preferred way of working only for short and non repetitive jobs. However, the real power of MATLAB for engineering calculations derives from its ability to execute a long sequence of commands stored in a file. Such files, which are generally called m-files since the filename has the form `filename.m`, may be either a function (see next laboratory class) or a script.

Although MATLAB provides its own editor, both scripts and functions are ordinary ASCII text files which can be created and edited using any text editor or word processor. A script is just a sequence of statements and function calls that could also be used at the MATLAB command prompt. It is invoked or 'run' by typing the filename (without the .m extension), and simply works through the sequence of statements in the script automatically.

Suppose that it is required to plot the function $y = \sin(\omega t)$ for different values of the variable ω (the frequency). A script called `plotsine.m` is created, as shown below. You can create it by using the MATLAB Editor: select the New (M-file) item on the MATLAB File Menu or click on the standard Microsoft Window's icon for a New Document, which is to be found near the top left of the MATLAB Command Window.

Type in the commands shown in the box below. Save your work to a file called `plotsine`. Note that the Matlab Editor will automatically add the .m file extension.

```
% This is a script to plot the function y = sin(omega*t).
%
% The value of omega (rad/s) must be in the workspace
% before invoking the script

t = [0: 0.01: 1];   % time
y = sin(omega*t);   % function output
plot(t,y)
xlabel('Time (s)'); ylabel('y')
title(' Plot of  y = sin(omega*t)')
grid
```

Important! At this stage, the commands in the box above should be saved in a text file. They should *not* be typed in the command window! Any problems, please consult a demonstrator. Similar applies to other boxed text in these notes.

Scripts should be well documented with comments, so that their purpose and functionality can be readily understood sometime after their creation. A comment begins with a %. Comments at the beginning of a script form a header which can be displayed using the `help` function. This is illustrated by the following example:

```
» help plotsine

This is a script to plot the function y = sin(omega*t).

The value of omega (rad/s) must be in the workspace
before invoking the script
```

If this help message does not appear, it may be because you have saved the file to a different location from the current MATLAB workspace. For example, if you saved the file to `h:\myfiles\plotsine.m` then change to this directory using `cd` before continuing:

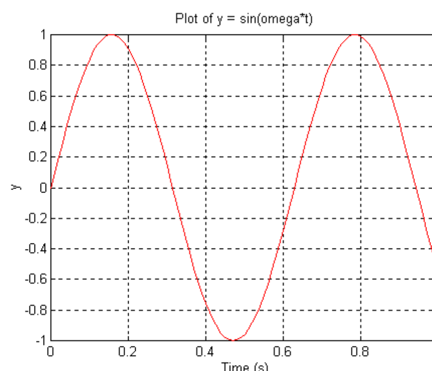
```
» cd h:\myfiles
» help plotsine
```

Next, type in a value for omega:

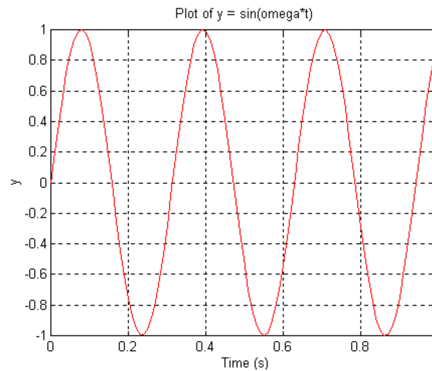
```
» omega=10
omega =
    10
```

Finally enter the name of your script in the command window and press return. If you wish to change the value of omega, you should 're-run' the script to update the plot, as shown below:

```
» plotsine
```




```
» omega=20
» plotsine
```



MATLAB program outputs can be made more informative by using the `disp` function. The purpose of this function is to display text or variable values on screen. Another useful function is `format compact`, which reduces the number of spaces that MATLAB inserts. The normal display is obtained by using the function `format loose`. Use of these functions is illustrated in the following example. Create a script for the following boxed text and save as an m-file called `example.m`

```
% Example Script
format compact
length = [5 10 3 2]; breadth = [1 5 2 0.5];
area = length.*breadth;
disp('      length   breadth   area')
disp('        m         m      sq m')
disp([length' breadth' area'])
```

The program is then run by typing its filename at the MATLAB command prompt:

```
» example
      length    breadth     area
        m          m        sq m
     5.0000     1.0000     5.0000
    10.0000     5.0000    50.0000
     3.0000     2.0000     6.0000
     2.0000     0.5000     1.0000
```

Note that the elements of the array variables `length`, `breadth` and `area` have been printed out as columns rather then rows. The apostrophe after the name changes the array from a row to a column, i.e. vector transpose (see Section 2.6 above). In fact, the expression `[length' breadth' area']` creates a matrix comprising these three columns.

1.3.1 For loops

A `for` loop allows a statement, or group of statements, to be repeated a fixed predetermined number of times. For example:

```
» for i = 1:5; x(i) = i*2; end
» x
x =
     2     4     6     8    10
```

In the above example, `i*2` is assigned to the elements of the array `x`, where `i` = 1 to 5. Notice that four statements have been written on one line, terminated by a semicolon to suppress repeated printing during the loop. The `x` at the end displays the final result. Note that each `for` statement must be matched with an `end` to close the loop.

1.3.2 Conditional statements

The MATLAB function **if** conditionally executes statements. The simple form is,

```
if expression
     statements
else
     statements
end
```

Both loops and conditional statements are most useful as part of a computer program – in other words, as a way of controlling what happens in a MATLAB script.

The following program illustrates both loops and conditional statements. The example is rather arbitrary, but study it carefully and experiment until you are confident about how these programming constructs work. The example is stored as the script file `example.m`

Notice that use has been made of the MATLAB function `size` which returns the number of rows and columns in the two dimensional array `x`. In this program, m = 1 since x is a single row, while n = 21. This saves the trouble of manually counting the number of elements in `x`.

```
%  This is a script to identify membership of the open
%  set {x:abs(x-3)<5} and the closed set {x:abs(x-3)<=5}

x = [-10:1:10]; [m,n] = size(x);
for i=1:n
  if abs(x(i) - 3) < 5
     yopen(i)=1;
  else
     yopen(i)=0;
  end
  if abs(x(i) - 3) <= 5
     yclosed(i)=1;
  else
     yclosed(i)=0;
  end
end
disp('     x   yopen yclosed');
disp([x' yopen' yclosed'])
```

```
» example
     x   yopen yclosed
   -10     0     0
    -9     0     0
    -8     0     0
    -7     0     0
    -6     0     0
    -5     0     0
    -4     0     0
    -3     0     0
    -2     0     1
    -1     1     1
     0     1     1
     1     1     1
     2     1     1
     3     1     1
     4     1     1
```

```
     5      1      1
     6      1      1
     7      1      1
     8      0      1
     9      0      0
    10      0      0
```

1.3.3 External data

MATLAB can import data in a number of formats, including simple text, spreadsheet files, image files, sound files and even movies. The `load` command is used to import ASCII text. Typically, such a file contains data collected from an engineering device or other system of interest. For example, use the Windows Notepad or the MATLAB Editor to create the following file called `test.txt` and save it to the PC hard drive or your network drive. Here, the first column might be the time, while the second column is a measurement, say a voltage. The data file should contain only numbers separated by commas or spaces:

```
10, 8
20, 11
30, 16
40, 15
50, 18
```

```
» load test.txt
» who
Your variables are:
  test
```

Note that the file extension `.dat` is required to load the data, but by default MATLAB assigns these data to a variable name that does *not* include the file extension. If you encounter a file-not-found error, then use the `cd` command to make sure that the working directory contains the data file in question. Alternatively, use the full path when loading the data, for example:

```
» load h:\myfiles\data\test.txt
```

1.3.4 Matrix Elements

The variable `test` above is a 5 by 2 matrix, i.e. 5 rows and 2 columns. Individual elements of this matrix may be extracted as illustrated below:

```
» test(3, 2)
ans =
      16
```

Recall that `ans` is the default variable name. As usual, you can assign the result to a named variable by using the equals symbol. Also, the colon character is used to assign an entire row or column:

```
» x = test(2, :)
x =
      [20  11]
```

The final example plots the second column against the first column:

```
» plot(test(:, 1), test(:, 2))
```

Exercise 5

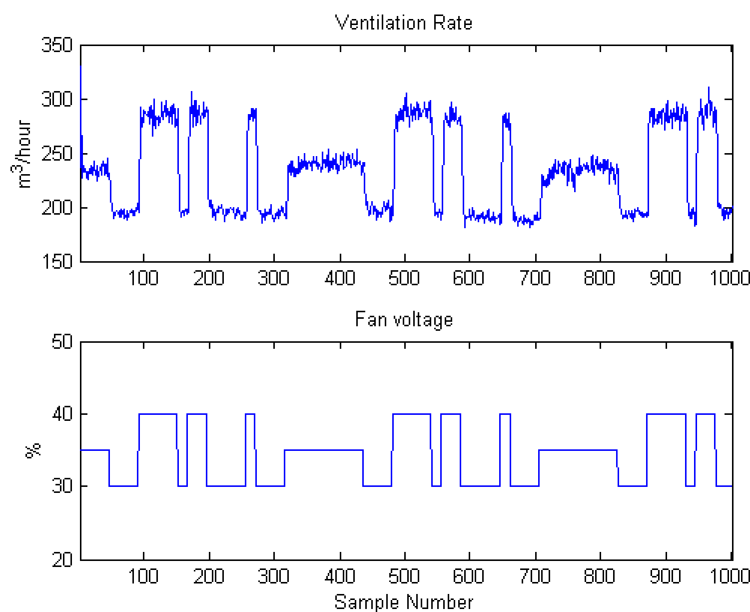
Consider the control of ventilation rate in an instrumented micro-climate test chamber. For these exercises, sample data files are provided on the Internet at:

http://www.lancs.ac.uk/staff/taylorcj/teaching/

Download one of these files to the PC hard drive or your network drive, e.g. when using some web browsers, right click and select **Save As…**

The first column of each data file is the ventilation rate (m^3/hour), while the second column is the input variable (fan voltage expressed as a percentage). The data are sampled every 2 seconds (each row of the file). Each file represents one particular experiment, showing how the ventilation rate responds to a given voltage signal.

1. Pick one of the data files. Write a script to load the data into MATLAB and generate a graph similar to the one shown below. You will need to use: `load`, `subplot`, `plot`, `title`, `ylabel`, `xlabel` and `axis`. Use the `help` command for more information about this new `axis` function: can you work out how to use it?



2. Use your script to very quickly plot data from the other files. If you have written an appropriate script for the first exercise above, all you have to do is change the filename. This is one advantage of using a script, as opposed to typing all the commands into the command window each time. If you have typed out the commands out one by one, or have used the menu system to annotate the graph, then it would be much more work!

Part 2 – Writing and using MATLAB functions

Both scripts and functions are text files with the extension .m and hence both are sometimes called MATLAB m-files. As discussed in Part 1, a *script* is a sequence of statements and function calls that could also be used at the MATLAB command prompt. It is invoked or 'run' by typing the filename and simply works through the sequence of statements in the script – just as though these were typed individually into the command window.

A *function* file differs from a script in that variables defined inside the file are local to the function and do not operate globally on the workspace: see examples below. For this reason, arguments may be passed from the workspace to the function. Similarly, any useful variables (outputs) from the function, should be defined as output arguments, or else they will not appear as variables in the workspace. This is similar to functions in other programming languages, such as 'C'.

2.1 Learning Objectives

This laboratory (Part 2) provides a basic introduction to using and writing functions in MATLAB. By the end of the session, you should be able to:

- understand the difference between global and local variables;
- use common built-in MATLAB functions such as `plot`, `cos` and `sin`;
- use pre-installed Toolboxes for extending the MATLAB language;
- use MATLAB functions with multiple input and output arguments;
- write your own functions to solve engineering problems.

Before leaving the class, check back on these objectives.

2.2 Built-in Functions

You should be familiar with several built-in functions, such as `plot`, `load` and `cos`. Such functions are 'hard-wired' into the software package and cannot be copied or modified.

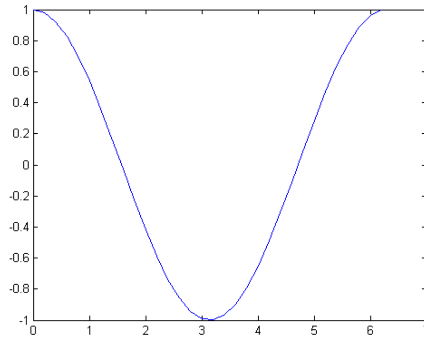
```
» cos(0)
ans =
       1
```

Here, the `cos` function takes the input argument zero and returns an output of unity, which is subsequently assigned to the default variable `ans`. The input and output arguments can be given any valid variable name. The function can also operate (element-by-element) on arrays:

```
» a = [0:0.2:2*pi]
a =
  Columns 1 through 8
         0    0.2000    0.4000    0.6000    0.8000 …
» b = cos(a)
b =
  Columns 1 through 8
    1.0000    0.9801    0.9211    0.8253    0.6967 …
```

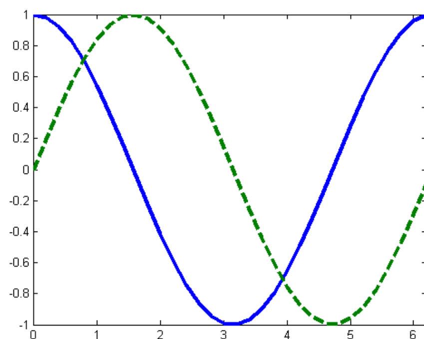
Some functions accept two or more input arguments, such as:

```
» plot(a, b)
```



In fact, `plot` can accept numerous input arguments, as the following example illustrates:

```
» c = sin(a)
» plot(a, b, '-', a, c, '--', 'linewidth', 2)
```



Here, two curves are plotted on the same graph (an alternative way of doing this is to use the `hold on` command), one with a solid trace, the other dashed. Finally, both lines are made thicker than the default case, as specified by the final two input arguments. More details about the plot function can be found by typing:

```
» help plot
```

Some functions return two or more output arguments:

```
» x = now
x =
  7.3291e+005
» [Y, MO, D, H, MI, S] = datevec(x)
Y =
        2006
MO =
           8
D =
        23
H =
        12
MI =
        13
S =
      4.9790
```

The example above first uses the `now` function to obtain a scalar representation of the current time and date: obviously this number will be different when you try this example! However, it is more useful to subsequently extract the year, months, date, hour, minutes and seconds from this scalar using `datevec`, which returns up to 6 output arguments.

Note that it is up to the user to specify any necessary output arguments. For example, the following call to `datevec` only returns the year and month:

```
» clear
» x = now
x =
  7.3291e+005
» [yr, mth]=datevec(x)
yr =
        2006
mth =
     8
» who
Your variables are:
mth  x    yr
```

Since the workspace was cleared at the start of this example, the day and time do not appear in the list of variables.

2.3 MATLAB Toolboxes

Type help and press return:

```
» help  <return>
HELP topics
matlab\general -  General purpose commands.
matlab\ops     -  Operators and special characters.
matlab\lang    -  Programming language constructs.
matlab\elmat   -  Elementary matrices and matrix manipulation.
matlab\elfun   -  Elementary math functions.
...
```

You will see a list of MATLAB toolboxes installed on your computer. Scroll up to see all the items on this list. Each toolbox contains various functions grouped together under a common heading. For example, typing `help elfun` provides a list of elementary math functions, including trigonometric functions.

```
» help elfun
  Elementary math functions.
  Trigonometric.
    sin         - Sine.
    sind        - Sine of argument in degrees.
    ...
```

Some of the functions in these toolboxes are very commonly used, others are more specialist in nature. For example, your version of MATLAB may include the Control System Toolbox, which includes numerous tools useful for the subject of control engineering. In fact, specialist

toolboxes are available for purchase (or sometimes free download) for a diverse range of other applications, including statistical analysis, financial modelling, image processing and so on.

Finally, note that most of these functions are *not* built-in. In other words, you can open, copy and even edit the function just as you would any other m-file. For example, `sind` listed in the example above can be opened in the MATLAB editor as follows:

```
» edit sind
```

However, it is best not to edit existing functions, since this is likely to corrupt your work if you later try to use these for their original purpose! Also, the next person to use the shared PC might get rather irate. Instead, it is more useful to write your own functions, as discussed next.

2.4 Writing MATLAB Functions

Functions are useful for extending the MATLAB language. Indeed, most users develop their own toolbox of personal functions. To illustrate this process, consider the next example.

2.4.1 Temperature Conversion Function

Create and save the file **`c2f.m`**

```
function f = c2f(c)
% Converts degrees Centigrade to degrees Fahrenheit
% Usage: f = c2f(c)

f = c*9/5 + 32;
```

Note that this file starts with the word `function`, to distinguish it from an ordinary MATLAB script. Use the `help` command to check that the file has been saved correctly.

```
» help c2f
Converts degrees Centigrade to degrees Fahrenheit
Usage: f = c2f(c)
```

If this help message does not appear, it may be because you have saved the file to a different location from the current MATLAB workspace. Change to this location using the `cd` command. The temperature conversion is obtained by invoking the function at the command prompt, just as for any other function:

```
» c2f(14)
ans =
      57.2000
```

In this example, an output argument was not specified, hence the answer is assigned to the default variable `ans` as usual – note that the output temperature is *not* called `f`. In fact, the variables `f` and `c` are both local to the function and do not exist in the workspace! You can check this by first clearing the workspace, as shown in the following example:

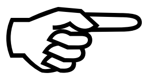
```
» clear
» cent = 14
cent =
      14
» fah = c2f(cent)
fah =
   57.2000
```

```
» who
Your variables are:
cent   fah
```

Here, the variables in the workspace are called `fah` and `cent` as would be expected. Finally, consider what happens if you introduce a variable called `c` into the workspace:

```
» c = 10000
» fah = c2f(cent)
fah =
   57.2000
» c
c =
   10000
```

Note that the workspace variable `c` is not effected by the `c2f` function in any way, even though a variable with the same name was used in the function.



Important! The variables in the workspace might have different names to those in the function file. If there are several input and output arguments, it is the order they appear in the first line of the function that matters *not their names*: see below.

2.4.2 Circle Function

The task is to draw a circle with centre (4,3) and radius 2. The first step is to create a function file called `circle.m` which will return the coordinates of a circle.

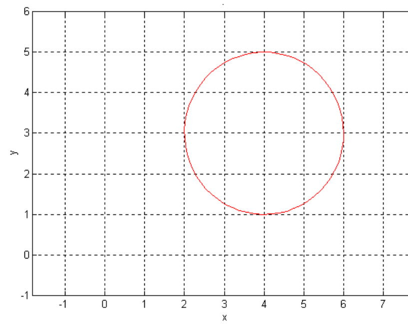
```
function [x,y] = circle(theta,a,b,r)
% Calling syntax: [x,y] = circle(theta,a,b,r)
% Returns the {x,y} coordinates corresponding to the
% angles theta, on the circle with centre {a,b} and radius r

x = a + r*cos(theta);
y = b + r*sin(theta);
```

This function is used to return the {x,y} coordinates corresponding to the angles theta on the circle. In the following script file called `example.m`, an array of angles is set up so as to compute the required {x,y} coordinates.

```
% Script to plot a circle
theta = [0:0.1:2*pi+0.1];
[x,y] = circle(theta,4,3,2);
plot(x,y)
axis([-1 7 -1 6]);
axis('equal');
xlabel('x')
ylabel('y')
grid on
```

This script is then invoked at the command prompt by typing its name in the usual way.



2.4.3 Global and Local Variables

It should be clear from the above examples that variables defined or modified in a function are local to that function. If you wish a variable to be transferred between the workspace and a function, then you should use input and output arguments. We also saw that function variables may have the same name as workspace variables, without any interaction between them.

By the workspace, we mean the normal operating environment in MATLAB. Variables defined using the command prompt or in a script, always appear in the workspace and can be listed using the `who` function. There is no distinction between the what is typed in the command window and in a script. In fact, any script can change the value of a variable in the workspace.

It is possible (although not often useful) to invoke a special command to make a variable into a global variable, as follows:

```
» global cent
```

Such a command can be typed in a script or in the command window. Once made global, the variable will be available for use in any function, without needing to explicitly pass it to that function as an input argument. Furthermore, if a function changes the value of the variable, then the change will immediately appear in the workspace.

However, good MATLAB programming practice will generally avoid the use of global variables. One advantage of writing functions is that, once they are thoroughly tested, you can reply on them to accurately complete a task. Global variables may interfere with the operation of a previously written function – or a function may inadvertently modify a global variable, causing ambiguities and confusion for the user.

2.4.4 Gas Charges Function

A gas company charges according to the (rather out of date and unrealistic) tariff below.

Quarterly usage (1000 units)	Standing charge (£)	Charge (£ per 1000 units)
0 to 20	200	80
20 to 50	600	60
50 to 100	1600	40
over 100	3600	20

The task is to create a function `gas.m` to calculate the charge for a given usage.

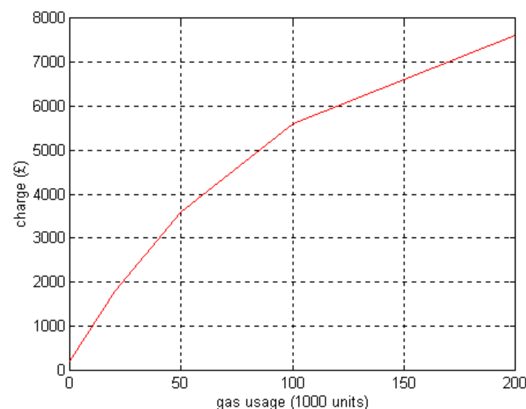
```
function charge = gas(usage)
% Returns the charge for the given usage of gas
[m,n] = size(usage); charge = zeros(m,n);
for i = 1:n
  if usage(i)<20
    charge(i)=200+80*usage(i);
  elseif usage(i)<50
    charge(i)=600+60*usage(i);
  elseif usage(i)<100
```

```
    charge(i)=1600+40*usage(i);
  else
    charge(i)=3600+20*usage(i);
  end
end
```

This piecewise linear function is used to return the set of charges for a specified usage. In the following script, an array `charge` is set up for this function, so as to compute the charges at the points of discontinuity. Type the name of the script to generate the graph.

Study the function and script carefully – make sure you know what is going on before continuing. Ask a demonstrator if you are unsure about anything.

```
% Example script
usage = [0:1:200];
charge = gas(usage);
plot(usage, charge)
xlabel('gas usage (1000 units)')
ylabel('charge (£)'); grid on
```



Exercise 6

Write a function with the following help message, to convert between Celsius and Fahrenheit temperature scales, with the direction of the conversation dependant on a 2nd input argument.

```
function t2 = c2f(t1, sw)
% Converts between degrees Fahrenheit and degrees Centigrade
% Usage: t2 = c2f(t1, sw)
%   t1: input temperature
%   t2: output temperature
%   sw: switch: if positive then function converts F -> C
%               if negative then function converts C -> F
```

2.5 Miscellaneous Functions

Before continuing, we will introduce several new functions and programming constructs that are sometimes useful for project work.

2.5.1 Complex Numbers and Roots

MATLAB allows for complex numbers, using *i* and *j* to represent the square root of minus one. Complex numbers can arise in the solution of quadratic equations, as illustrated in the following example. Create and save a function called `quadeq.m`

```
function [x1,x2] = quadeq(a,b,c)
% Solves ax^2 + bx + c = 0
y = sqrt(b^2 - 4*a*c);
x1 = (-b + y)/(2*a);
x2 = (-b - y)/(2*a);
```

This function can then be used at the MATLAB command prompt:

```
» [x1,x2]=quadeq(1,2,3)
x1 =  -1.0000 + 1.4142i
x2 =  -1.0000 - 1.4142i
```

Notice that `y` in the function `quadeq` may be either real, imaginary or zero. If `y` is imaginary then `x1` and `x2` will be complex as above, otherwise `x1` and `x2` will be real:

```
» [x1,x2]=quadeq(1,-3,2)
x1 =   2
x2 =   1
```

In fact, MATLAB already has a function `roots` for finding the roots of any polynomial. The input argument is a one dimensional array containing the coefficients of the polynomial:

$$c_1x^n + c_2x^{n-1} + ... + c_nx + c_{n+1}$$

Conversely, the coefficients of a polynomial can be obtained from the roots by using the function `poly`. To illustrate these functions, consider the following example:

$$x^3 - 3x^2 + 6x - 4$$

At the MATLAB command prompt, the coefficients of the polynomial are entered as the array `c1` and the function `roots` is invoked.

```
» c1 = [1 -3 6 -4];
» r = roots(c1)
r =
   1.0000 + 1.7321i
   1.0000 - 1.7321i
   1.0000
```

If the function `poly` is now invoked with the array `r` as the argument, the coefficients of the original polynomial are obtained.

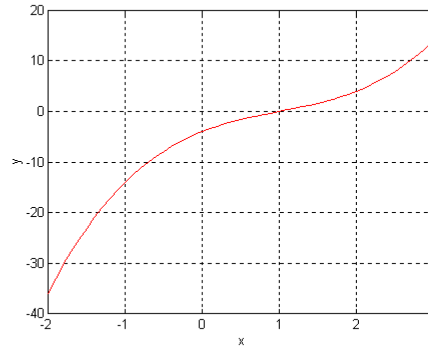
```
» c2 = poly(r)
c2 =
    1.0000   -3.0000    6.0000   -4.0000
```

This function is more useful in cases where you know the roots and wish to find the unknown polynomial – here we are just testing out the functions!

Finally, given the coefficients, the polynomial function can be plotted over the range *x* using the function `polyval`. A script file for this purpose is shown below.

```
% Example script
c = [1 -3 6 -4];
x = [-2:0.1:3];
y = polyval(c,x);
plot(x, y); xlabel('x'); ylabel('y'); grid on
```

Invoking this script file at the MATLAB command prompt will cause the graph to be plotted.



In this context, it is useful to briefly mention two more MATLAB functions, `real` and `imag`, which are used to return the real and complex parts of an argument respectively.

2.5.2 Storing data

Arrays of data can be stored as ASCII files for use by another program using the `save` command, as shown below:

```
z=[1 2 3 4; 5 6 7 8]
save data1.dat z /ascii
```

Here, each row of the array `z` will be written to a separate line in the data file. Alternatively, the data file may be created using a word processor or an editor. The information contained in a data file can be read by a MATLAB program using the `load` command.

2.5.3 Formatted Output

The `fprintf` command gives more control over printed output than the `disp` command used so far. The general form of this command is as follows:

```
fprintf(string, variables)
```

where `string` contains text and format specifications enclosed in single quotation marks and `variables` is a list of arrays to be printed. The format specifiers `%e`, `%f`, and `%g` are used to place the variables in the text and to define the number format as follows,

`%e`	exponential notation
`%f`	fixed decimal point notation
`%g`	either of the above depending upon which is shorter

A newline is specified by the string `\n`. Simple examples are shown below:

```
» temp = 78;
» fprintf('The temperature is %f degrees F \n',temp)
The temperature is 78.000000 degrees F
» fprintf('The temperature is \n %f degrees F \n',temp)
The temperature is
 78.000000 degrees F
```

The format specifiers can also be used to control the number of decimal places and the number of characters for printing numbers, as shown below,

```
» fprintf('The temperature is  %4.1f degrees F \n',temp)
The temperature is  78.0 degrees F
```

In this example, the value of `temp` is printed with 4 characters, including the decimal point and one decimal place. Note that `fprintf` is most useful when called from within scripts and functions.

2.5.4 While Loops

The general format for a while loop is as follows:

```
while expression
     statements
end
```

If the expression is true then the statements are executed. After the statements are executed the condition is retested. If the condition is still true the statements are executed again. When the condition is false the program skips to the statements following the `end` of the while loop.

```
» k = 1;
» while k <= 4
     fprintf('Time #%2.0f: Hello world \n',k);
     k = k + 1;
  end
Time # 1: Hello world
Time # 2: Hello world
Time # 3: Hello world
Time # 4: Hello world
```

2.5.5 Logical expressions

Six relational operators can be used to compare values in logical expressions:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to

Note the distinction between a single equals sign (assign a value to a variable) and a double equals sign (comparison). For example:

```
» a = 2
a =
    2
» a == 3
ans =
     0
» a == 2
ans =
     1
```

```
» a >= 0
ans =
      1
```

As can be seen above, if the variables to be compared are scalars the result is 1 if true or 0 if false. If the variables to be compared are arrays, they must be of the same dimension. The corresponding array elements are compared and the result is an array of the same dimension with 0's and 1's as its elements, according to the result of the element by element comparison. This is illustrated in the following example:

```
» a = [2 4 6];
» b = [3 5 1];
» a<b
ans =
     1     1     0
» a~=b
ans =
     1     1     1
```

Logical expressions can be preceded by the logical operator not ~ to change the value of the logical expression to the opposite value.

```
» a = [2 4 6];
» b = [3 5 1];
» ~(a<b)
ans =
     0     0     1
```

MATLAB has some useful logical functions. Two of these are:

`any(x)` For each column of the array x this function returns 1 (true) if any elements of the array are nonzero, or 0 (false) otherwise.

`all(x)` For each column of the array x this function returns 1 (true) if all the elements of the array are nonzero, or 0 (false) otherwise.

These functions are illustrated below:

```
» b = [1 0 4
       0 0 3
       8 7 0];
» any(b)
ans =
     1     1     1
» all(any(b))
ans =
     1
» any(all(b))
ans =
     0
```

2.6 Engineering Problem Solving

A systematic five step approach to engineering problem solving using MATLAB is suggested below:

A. Problem Statement. A clear, concise statement of the problem.

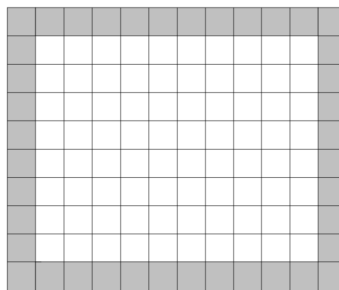
B. Input/Output Description. The given information and the information to be computed.

C. Hand Example. Solve the problem by hand using a simple set of data.

D. MATLAB solution. Use the appropriate MATLAB commands to obtain the solution.

E. Test the solution. Using a variety of data sets.

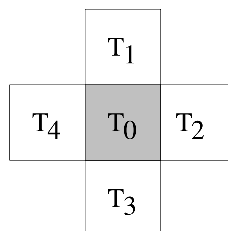
Consider the temperature distribution of a rectangular plate in which the temperatures of three sides are maintained at a constant value and the temperature of the remaining side is maintained at a different value.



Imagine a rectangular grid drawn over the plate as shown above. The problem is to compute the temperature of each rectangular section of the plate, given the temperatures of the shaded sections. One numerical procedure, known as relaxation, is to assume values for the unknown temperatures of the sections within the plate and to compute a new temperature for each of the unshaded sections in turn using the formula:

$$T_0 = \frac{T_1 + T_2 + T_3 + T_4}{4}$$

where T_0 is the new temperature of a section and T_1, T_2, T_3 and T_4 are the current temperatures of the adjacent sections as indicated in the diagram below:



This process is carried on repeatedly, continually sweeping across and down the plate covering all sections until the calculated temperatures cease to alter by a specified (small) amount called the tolerance. This is an example of iteration, a commonly used numerical technique of repeated calculation until a sufficiently accurate result is obtained. Of course, the particular formula to be used depends upon the particular problem to be solved. For this example, the 5 steps above take the following form:

A. Problem statement

Determine the equilibrium temperature distribution for a rectangular plate with fixed temperatures on its sides.

B. Input / output description

The input is the size of the grid, the temperatures of the sides, and the iteration tolerance. The output is the set of temperature values for the sections of the plate.

C. Hand example

To be sure that the process is understood, carry out two iterations using a coarse grid. Assume four rows and four columns, that the top, left and right hand sides are at 100 degrees and that the bottom is at 50 degrees. Take the initial values of the temperatures to be computed as 0 degrees. Common sense indicates the closer the initial guess is to the actual values, the fewer the number of iterations required to achieve the desired accuracy. However, the ultimate success of the procedure should not be dependent on the initial values.

Initial Temperatures:

```
   100   100   100   100
   100     0     0   100
   100     0     0   100
    50    50    50    50
```

First iteration:

```
   100   100   100     100
   100    50    50     100
   100    37.5  37.50  100
    50    50    50      50
```

Note that the biggest change is 50 - 0 = 50. Second iteration:

```
   100   100   100     100
   100    71.9  71.9   100
   100    59.4  59.4   100
    50    50    50      50
```

Note that the biggest change is 71.875 - 50 = 21.875. The solution is starting to converge.

D. MATLAB solution

The program is written and stored as script `example.m`

```
%     This program initializes the temperatures in a
%     metal plate and determines the equilibrium
%     temperatures based on a tolerance value.
%
%     Program adapted from Engineering Problem Solving
%     with MATLAB by D M Etter, Prentice Hall, 1993
nrows = input('Enter number of rows ');
ncols = input('Enter number of columns ');
iso1 = input('Enter temperature for top and sides ');
iso2 = input('Enter temperature for bottom ');
tolerance = input('Enter equilibrium tolerance ');
%
%     Initialize and print temperature array.
%
old = zeros(nrows,ncols);
old(1,:) = iso1 + zeros(1,ncols);
old(:,1) = iso1 + zeros(nrows,1);
```

```
old(:,ncols) = iso1 + zeros(nrows,1);
old(nrows,:) = iso2 + zeros(1,ncols);
disp('Initial Temperatures');
disp(old)
new = old;
equilibrium = 0;    % logical false
%
%     Update temperatures and test for equilibrium.
%
while ~equilibrium  %  not false = true
   for m = 2:nrows-1
      for n = 2:ncols-1
         new(m,n) = (old(m-1,n) + old(m,n-1) +...
                     old(m,n+1) + old(m+1,n))/4;
      end
   end
   if all(new-old <= tolerance)    % check all changes
      equilibrium = 1;             % logical true
      disp('Equilibrium Temperatures');
      disp(new)
    end
    old = new;
end
```

E. Test the program

Repeat the calculation using the script, i.e. type the scripts name at the MATLAB prompt.

```
» example
Enter number of rows 4
Enter number of columns 4
Enter temperature for top and sides 100
Enter temperature for bottom 50
Enter equilibrium tolerance 40
Initial Temperatures
   100   100   100   100
   100     0     0   100
   100     0     0   100
    50    50    50    50
Equilibrium Temperatures
  100.0000  100.0000  100.0000  100.0000
  100.0000   71.8750   71.8750  100.0000
  100.0000   59.3750   59.3750  100.0000
   50.0000   50.0000   50.0000   50.0000
```

Since the tolerance was set at 40, the second iteration is accepted. A more realistic result is obtained with a smaller tolerance and smaller mesh size (more sections), as shown below:

```
» example
Enter number of rows 6
Enter number of columns 6
Enter temperature for top and sides 100
Enter temperature for bottom 50
```

```
Enter equilibrium tolerance 1
Initial Temperatures
   100   100   100   100   100   100
   100     0     0     0     0   100
   100     0     0     0     0   100
   100     0     0     0     0   100
   100     0     0     0     0   100
    50    50    50    50    50    50
Equilibrium Temperatures
  100.0000  100.0000  100.0000  100.0000  100.0000  100.0000
  100.0000   96.1837   93.9040   93.9040   96.1837  100.0000
  100.0000   92.0108   87.4372   87.4372   92.0108  100.0000
  100.0000   86.3297   79.4841   79.4841   86.3297  100.0000
  100.0000   75.7305   67.7698   67.7698   75.7305  100.0000
   50.0000   50.0000   50.0000   50.0000   50.0000   50.0000
```

Exercise 7

Modify the program above so that each of the four sides can have different temperatures and so that the number of iterations is printed out.

Part 3 – Simulation of 1st and 2nd order Transfer Functions

In order to understand the behaviour of mechanical and electrical systems the laws of physics are used to set up mathematical models. For example, if the system is dynamical (i.e. the output changes gradually over time) then continuous-time mathematical models will generally be in the form of differential equations, in which the independent variable is time.

The solution of a set of differential equations which make up the mathematical model will provide the behaviour of the system through time: e.g. the speed of response; the final value of the output following a step input; the presence of any oscillations; and so on. Numerical procedures are well established for the solution of a very wide class of linear and nonlinear differential equations. Of course, these are usually solved on a computer.

In this regard, SIMULINK™ is an advanced 'iconographic' computer simulation system for general, nonlinear, dynamic systems that combines a block diagram interface and simulation capabilities, with the core functionality of MATLAB™. In particular, SIMULINK is based around a block diagram library, where icons representing simulation elements can be built into blocks that are then arranged in a window on the computer screen, and connected by lines that carry the variables used in the equations.

The user develops a simulation by using the mouse to drag standard blocks from the SIMULINK library into the model window. Such pre-built blocks include, for example: a step input function; integrators; both continuous and discrete-time transfer function models; graphing functions; and various nonlinear elements such as saturation, time delay or dead-zone blocks. SIMULINK can be used to develop both linear and nonlinear simulation models.

3.1 Learning Objectives

This laboratory (Part 3) is an introduction to SIMULINK. The discussion is limited to continuous-time systems represented by differential equations.

Note that transfer function models and time constants etc. are studied in some Engineering Department taught modules. However, if you are using these notes only as an introduction to the software package, the mathematical background can be probably be ignored. Otherwise, by the end of the session, you should be able to:

- develop SIMULINK models for 1st and 2nd order transfer function models;
- have a basic understanding how the time constant, steady state gain, damping ratio and natural frequency relate to the time response.
- graph the results using the `Scope` block in SIMULINK; and save the results to the MATLAB workspace for graphing using the `plot` function.

Before leaving the class, check back on these objectives.

3.2 Plotting a Sine Wave using the Scope

This section describes how to drag icons from the SIMULINK block library and how to use the `Scope` for plotting graphs.

Note that different versions of SIMULINK may operate slightly differently from the description in these notes – try experimenting to overcome any such problems, or ask a demonstrator.

Step 1. Start MATLAB. Click on its desktop icon or use the Windows Start menu to begin a new MATLAB session. A 'splash' screen will temporarily appear before MATLAB eventually starts up. Find the *Matlab Command Window*: here you can type commands at the >> prompt. Start up the SIMULINK package by typing 'simulink' and pressing return, as shown below:

```
MATLAB Command Window

This is a Classroom License for instructional use only.
Research and commercial use is prohibited.
To get started, select "MATLAB Help" from the Help menu.

>> simulink          <press return>
```

The *Simulink Library Browser* window should appear on screen, with a list of subdirectories perhaps including ‘Simulink’, ‘Control System Toolbox’, ‘S-Function Demos’ and so on. From now on, these notes will refer to this window as simply the *Simulink Library*. If at any point you accidentally close it, type ‘simulink’ again to restart.

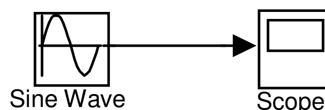
The SIMULINK library consists of numerous ready made simulation blocks for you to use, such as transfer functions, signal generators and graphing functions. The library is usually arranged hierarchically into subgroups opened and closed by clicking on the appropriate icon.

Step 2. Create a new model. To do this, click on the usual Microsoft Window’s icon for a New Document, which is to be found near the top left of the *Simulink Library*. An empty window will appear: this is where you are going to build your block diagram. From now on, these notes will refer to this window as your *model*.

Step 3. Copy the block for a Sine Wave onto your empty model window. To do this, first click on the + symbol next to the word ‘Simulink’ at the top of the *Simulink Library*, to produce a list of subdirectories called ‘Continuous’, ‘Discrete’, ‘Look-Up Tables’ and so on. Next, click on the ‘Sources’ icon to open up a list of ready made blocks including: ‘Band-Limited White Noise’, ‘Chirp Signal’, ‘Clock’, ‘Constant’ etc. Scroll down this list until you find the block called ‘Sine Wave’. Use the mouse to drag the Sine Wave icon onto your model window. In these notes, we might later abbreviate the location of the ‘Sine Wave’ block as follows: Simulink>Sources>Sine Wave.

Step 4. Copy the Scope block onto your diagram. In a similar manner to step 3 above, find and copy the block labelled ‘Scope’ onto your model. This block may be found in the ‘Sinks’ subdirectory, i.e. Simulink>Sinks>Scope. As we will see below, the Scope is used to plot simulation results.

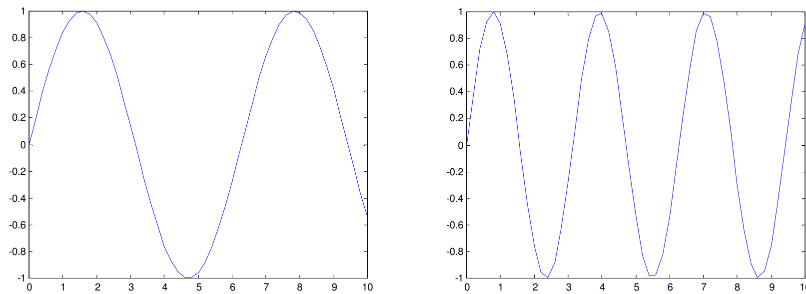
Step 5. Connect a line from the Sine Wave to the Scope. On your model window, drag a line from one block to the other. A little practice should allow you to learn how to connect up SIMULINK block diagrams fairly easily. Hopefully your model now looks like this:



Tip: If you get into a mess, the Delete key can be used to remove selected blocks or lines. Also, you can reposition blocks by dragging them with the mouse.

Step 6. Simulation results. Double click on the Scope block to open up a plot window with an empty black graph. Next, run (or simulate) your block diagram by selecting the ‘Start’ option from its ‘Simulation’ menu. Alternatively, click on the ‘Start/Pause Simulation’ icon, which looks like the play button of a video recorder. Hopefully, a graph of the sine wave will appear on the Scope window, looking something like the left hand plot below. If the graph is now behind some other windows, then find it and bring it to the front of the screen.

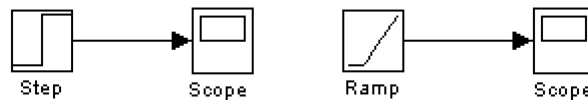
Tip: On the Scope Window, click the binoculars icon to scale the graph to fill the plot.



Step 7. Try different types of Sine Wave. The settings for the sine wave, such as its frequency and magnitude, can be changed by double clicking on the Sine Wave block on your model. This opens up a 'Block parameters' dialogue box. Try this now, changing the frequency from 1 to 2 radians per second. To see the results, you have to run the simulation again, i.e. click on the play button or select 'Start' as in step 6 above. *You always have to simulate the system again in order to update the results.*

Exercise 8

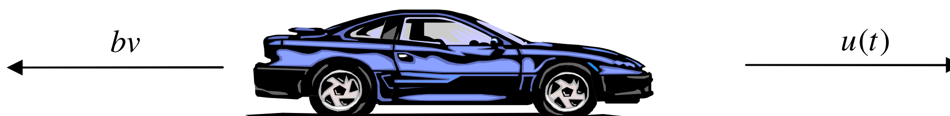
Use the `Scope` to plot graphs for the following blocks: Simulink> Sources> Step and Ramp.



Tip: You can remove the Sine Wave from your block diagram by single clicking on its icon (so that it is highlighted by little black squares in each corner) and pressing Delete on your keyboard. Drag and drop the other input blocks from the *Simulink Library* to replace it. Alternatively, you may create a new block diagram from scratch or even have all three in the same diagram, in which case you will have three separate scope blocks to double click on. If you wish, you may save any of your models to your network directory or a removable disk using the standard Microsoft Windows 'Save' option from the 'File' menu.

3.3 Vehicle Speed – 1st Order System

The example below is based on the mathematical model for vehicle speed developed during a Part I module in Engineering, namely: ENGR. 115 Computers & Control. However, a brief review of the approach is also given below.



We will develop a linear continuous-time model for the speed of a vehicle with mass m (kg). Denote the velocity of the vehicle v (m/s). The rate of change of the velocity is the acceleration $\dot{v}$ (m/s^2). Assume that the engine imparts a force $u(t)$. Assume that friction slows the car down and is proportional to the velocity, i.e. friction force equals bv where b is a coefficient, as shown in the diagram above. Finally, assume that the rotational inertia of the wheels is negligible. Using Newton's Law $F = ma$, we can write the net force:

$$u(t) - bv = m\dot{v} \qquad (1)$$

By rearranging the equation, we can treat engine force $u(t)$ as an input (right hand side) and the velocity v as an output (left hand side):

$$m\dot{v} + bv = u(t) \qquad (2)$$

Finally, we can divide through by b as follows:

$$\frac{m}{b}\dot{v} + v = \frac{1}{b}u(t) \qquad (3)$$

This is a 1st order differential equation. Compare with the general 1st order equation:

$$\tau\,\dot{x} + x = Ku(t) \qquad (4)$$

where x is the *output* variable (speed), $u(t)$ is the *input* variable (engine force), τ is the *time constant* and K is the *steady state gain*. Comparing equations (3) and (4), it is clear that the time constant and steady state gain of the vehicle speed model are given by:

$$\tau = \frac{m}{b} \quad ; \quad K = \frac{1}{b} \qquad (5)$$

3.3.1 Transfer Function for Vehicle Speed

One way to represent equation (4) is to develop a Transfer Function model. Briefly, the idea is to use an 'operator' notation (more formally, this closely relates to taking Laplace Transforms and assuming zero initial conditions), as follows:

$$\dot{x} = \frac{dx}{dt} = sx\,, \qquad \ddot{x} = \frac{d^2x}{dt} = s^2x\,, \qquad \ldots, \qquad \frac{d^nx}{dt} = s^nx \qquad (6)$$

In this case, the 1st order differential equation (4) can be denoted as follows:

$$\tau\,\dot{x} + x = Ku \quad \rightarrow \quad \tau\,sx + x = Ku \qquad (7)$$

Rearranging yields a Transfer Function relating the input *u* to the output *x*:

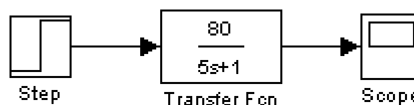
$$(\tau\,s + 1)x = Ku \quad \rightarrow \quad x = \frac{K}{\tau\,s + 1}u \qquad (8)$$

As an example, if we assume $\tau = 5$ and $K = 80$, then the model becomes:

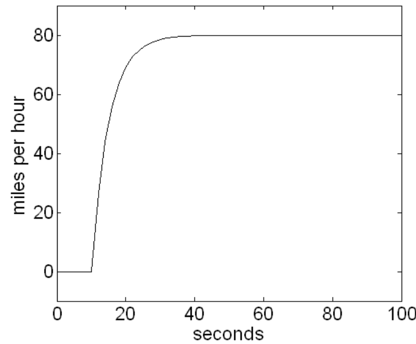
$$x = \frac{80}{5s + 1}u \qquad (9)$$

<u>Exercise 9</u>

Create the block diagram shown below, by dragging and dropping the appropriate blocks from the *Simulink Library* and entering the required parameter values. In SIMULINK the numerator is a scalar [80], while the denominator is represented by a vector [5, 1] or simply [5 1].



Graph the results in the `Scope` for different values of τ and K. Consider how the shape of the response depends of the values of τ and K.



Exercise 10

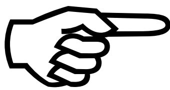
Reproduce the graph above (for $\tau = 5$ and $K = 80$) by saving the SIMULINK data into MATLAB and then using the `plot` command. Detailed instructions follow:

- SIMULINK: double click on the `Scope` block; click on the 'parameters' icon; select 'data history' tab; enter a variable name (e.g. '`x`'); select 'Array' from the 'Format' drop down menu; run the SIMULINK simulation again (i.e. click the 'Start' button).
- MATLAB command window: type the command `plot(x(:, 1), x(:, 2))` i.e. to graph the 2nd column (speed) against the 1st column (time). Annotate the graph as shown in the first laboratory class (202.P1 Exercise 5).

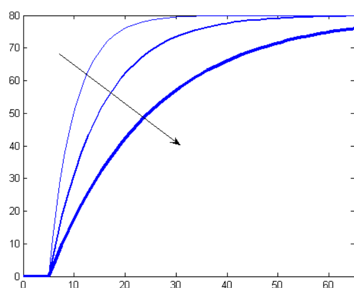
The advantage of graphing the results using MATLAB is that the figure can be properly labelled with the axis titles and the legend, as required for formal engineering reports.

Exercise 11

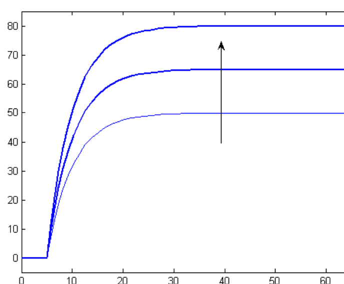
Repeat the above experiments for the Ramp and Sine Wave inputs. Again, how does the shape of the response depend of the values of τ and K?



Important! If you are studying control engineering, we will look at the theoretical reasons for the various dynamic responses in the lecture course, so it is important to make a note of all these results. Either make brief written notes with sketches, or save a few example graphs for your own later reference. For the step response, my own notes look like this:



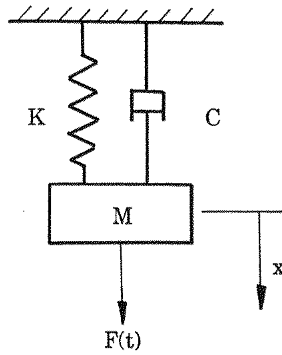
larger τ = slower response



larger K = larger steady state output
(in fact, steady state output = K * input)

3.4 Mass Spring Damper – 2nd Order System

The mass-spring-damper shown below is a classic example of a 2nd order system. Here, M is the mass, K is the spring stiffness, C is a parameter for the damping, $F(t)$ is an external force and x is the displacement (measured in metres from an arbitrary reference point).



This example will be studied in more detail during the ENGR.202 lectures. However, in this laboratory class, the focus is on implementing the model using SIMULINK. By making a number of assumptions, we can derive the following mathematical model (left hand side):

$$\frac{d^2x}{dt^2} + \frac{C}{M}\frac{dx}{dt} + \frac{K}{M}x = \frac{1}{M}F(t) \qquad \text{OR} \qquad \frac{d^2x}{dt^2} + 2\zeta\omega_n\frac{dx}{dt} + \omega_n{}^2 x = ku(t)$$

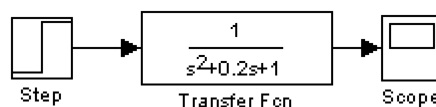
The model compares with the general form of a 2nd order differential equation, which is shown above right. In this example, the output variable *x* is the displacement of the mass and the input variable *u* is the force $F(t)$. The **damping ratio** (ζ) and **natural frequency** (ω_n) are functions of the parameters *M*, *K* and *C*, which may vary from system to system. One way to solve this equation is use the 'operator' notation introduced on page 4, as shown below:

$$s^2x + \frac{C}{M}sx + \frac{K}{M}x = \frac{1}{M}u \quad \rightarrow \quad \left(s^2 + \frac{C}{M}s + \frac{K}{M}\right)x = \frac{1}{M}u \quad \rightarrow \quad x = \frac{\frac{1}{M}}{s^2 + \frac{C}{M}s + \frac{K}{M}}u$$

Given an input signal *u*, the Transfer Function above can be used to find the behaviour of the output variable *x* over time. The following exercise looks at the unit step response.

Exercise 12

Step 1. Create the block diagram shown below. Here, assume that the mass $M = 1$, the damping $C = 0.2$ and the spring stiffness $K = 1$. To program these parameters into SIMULINK, double click on the Transfer Fnc block and change the settings in the dialogue box so that the numerator is [1] and the denominator is the vector [1, 0.2, 1].

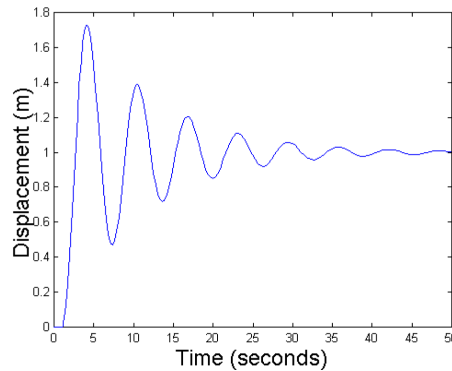


Step 2. This system takes about 50 seconds to reach steady state, so you need to change the simulation length from the default 10 seconds. From the 'Simulation' menu of your model, select 'Configuration Parameters' to open a dialogue box. Change 'Stop time' from 10 to 50.

Step 3. The default sample size for the numerical integration routine is too course (at least in the version of SIMULINK used to prepare these notes) so you should change it as follows: from the 'Simulation' menu, select 'Configuration Parameters' and set both the 'Relative tolerance' and 'Absolute tolerance' to 1e-6.

Step 4. Now run the simulation as usual. Remember to double click on the `Scope` block to plot the results and then click on the *binoculars icon* to zoom the graph. The model represents

the mass on the end of the spring bouncing up and down, with the oscillations decaying gradually over time, as shown in the figure below (you do not need to annotate your figure).



Step 5. The *characteristic equation* of a transfer function is obtained by setting its denominator equal to zero. For the mass-spring-damper system with $M = 1$, $C = 0.2$ and $K = 1$, the characteristic equation is:

$$s^2 + 0.2s + 1 = 0$$

The roots of this equation, called the *poles*, define the stability and time response of the system. It is straightforward to find these pole using MATLAB as shown below:

```
» roots([1, 0.2, 1])
ans
  -0.1000 + 0.9950i
  -0.1000 - 0.9950i
```

Here, i represents the complex number. Note that $s^2 + 0.2s + 1$ is entered into the MATLAB Command Window in just the same way as in SIMULINK, i.e. the coefficients are listed in descending powers of s enclosed in square brackets. Since these are used as an input argument to the function `roots`, the square brackets are then surrounded by round brackets as usual.

Technical comment on these results: In this example, the two poles both have negative real components (-0.1) so the system is stable, i.e. the oscillations gradually decay to zero and the displacement x settles down to a constant (steady state) level. In addition, the complex component ($+/- \; 0.995\,i$) ensures that the response is oscillatory, as we have seen above.

Exercise 13

This exercise examines the step response of the system for various damping parameters. Using the same model from Exercise 5, double click on the Transfer Fnc block to change the damping parameter from $C = 0.2$ to each of the following values: $C = 5$, $C = 0.05$, $C = 0$ and $C = -0.05$. When $C = 0$, make sure that you include the nought, i.e. the denominator coefficients for this case should be `[1, 0, 1]`.

For each case:

- Find the poles using MATLAB.
- Examine the time response using SIMULINK. Don't forget to zoom in on the `Scope` using the binoculars button after each simulation.

How does the parameter C and the associated poles affect the damping and stability?

What is your physical interpretation of these results for a mass-spring-damper system?

Can you think of any real system with similar dynamics?

3.5 Natural and Damped Frequencies

The transfer function of a general 2nd order system is as follows:

$$x = \frac{k}{s^2 + 2\zeta\omega_n s + \omega_n^2} u \qquad (10)$$

In the case of the mass-spring-damper above, the natural frequency $\omega_n = 1$, the damping ratio $\zeta = 0.1$ and the gain $k = 1$, i.e.,

$$x = \frac{1}{s^2 + 0.2s + 1} u \qquad (11)$$

The damping ratio ζ determines how quickly the oscillations decay or grow over time: see Exercise 6 above. The *frequency* at which a system actually oscillates is called the damped frequency ω_d and can be found from the period of oscillations *T* as follows:

$$\omega_d = \frac{2\pi}{T} \quad \text{(radians per second)} \qquad (12)$$

In fact, ω_d is related to the natural frequency and damping ratio as follows: $\omega_d = \omega_n \sqrt{1-\zeta^2}$

When the damping ratio is zero ($\zeta = 0$), the system is said to be critically damped and the oscillations will (unrealistically) continue forever. In fact, we can see from the equation above that when $\zeta = 0$, then $\omega_d = \omega_n \sqrt{1-0} = \omega_n$, i.e. the natural frequency is equal to the damped frequency. This property is used in the exercise below.

Exercise 14

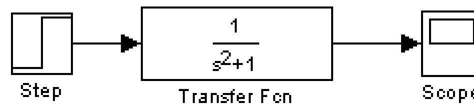
For the transfer function model of a general 2nd order system (10), set $\zeta = 0$ and $k = 1$ so that the system oscillates at its natural frequency, i.e.,

$$x = \frac{k}{s^2 + 2\zeta\omega_n s + \omega_n^2} u \quad \rightarrow \quad x = \frac{1}{s^2 + \omega_n^2} u$$

For the first example, assume that $\omega_d = \omega_n = 1$. In this case, the period of oscillations can be determined in advance by rearranging equation (12) as follows:

$$T = \frac{2\pi}{\omega_d} = 2\pi = 6.28 \text{ seconds}$$

Step 1. Program the above model for the case that $\omega_n = 1$ and examine the response using the Scope and binoculars button as usual. The denominator should be: `[1, 0, 1]`.



Step 2: Use the 'Zoom X-axis' button on the Scope to zoom in and measure the period of oscillations, e.g. the time from one peak (or trough) to the next. It should be 6.28 seconds!

Step 3: Try these frequencies: $\omega_n = 2$, $\omega_n = 0.5$ and $\omega_n = 0.25$. Note that ω_n should be squared in the dialogue box, e.g. for $\omega_n = 2$, the denominator is: `[1, 0, 4]`.

Step 4: Repeat for $\zeta = 0.1$, together with $\omega_n = 1$, $\omega_n = 2$, $\omega_n = 0.5$ and $\omega_n = 0.25$.

How does the natural frequency ω_n affect the step response of a second order system?

Part 4 – Linear and Nonlinear Differential Equations

Part 3 used SIMULINK™ to solve 1st and 2nd order linear differential equations by representing them as Transfer Function models. This approach is taken a step further in the present laboratory by examining the general *n*th order linear differential equation. However, the real power of SIMULINK comes from its ability to solve more complicated models built up from Integrator blocks, coupled with nonlinear elements such as saturations, time delays or products, as discussed later.

4.1 Learning Objectives

This laboratory (Part 4) continues the introduction to SIMULINK. As for Part 3, the discussion is limited to continuous-time systems. By the end of the session, you should be able to:

- develop SIMULINK models for *n*th order Transfer Function models;
- use common built-in SIMULINK blocks such as the `Step`, `Sum` and `Integrator`;
- develop SIMULINK models for linear and nonlinear systems using integrators.

Before leaving the class, check back on these objectives.

4.2 Differential Equations solved using Transfer Functions

Physical systems can often be modelled by 1st or 2nd order linear differential equations. However, many systems encountered in engineering are more complex and require higher order differentials to represent their dynamic behaviour. Furthermore, once we introduce controllers into the model, the order of the overall system will be even higher.

A general *n*th order differential equation is shown below:

$$a_0 \frac{d^n x}{dt^n} + a_1 \frac{d^{n-1} x}{dt^{n-1}} + \ldots + a_{n-1} \frac{dx}{dt} + a_n x \quad = \quad b_0 \frac{d^m u}{dt^m} + b_1 \frac{d^{m-1} u}{dt^{m-1}} + \ldots + b_{m-1} \frac{du}{dt} + b_m$$

where x is the output variable (e.g. the displacement of the mass-spring-damper) and u is the input variable (e.g. the force), while $a_0 \ldots a_n$ and $b_0 \ldots b_m$ are the model parameters. Note that in addition to differentials of the output variable x, the input signal u also has differential terms. The system above may be written in Transfer Function form as follows:

$$x = \frac{b_0 s^m + b_1 s^{m-1} + \cdots + b_m}{a_0 s^m + a_1 s^{m-1} + \cdots + a_m} u$$

Although these equations may look complicated and are time consuming to solve by hand using Laplace Transforms, they are very straightforward to simulate using SIMULINK. The key point to remember is that you enter coefficients of the Transfer Function in descending powers of s, separated by commas (or spaces) and surrounded by square brackets. For example, the handout for Practical 3 covered simulation of the following 2nd order system (e.g. mass-spring-damper) based on $n = 2$ and $m = 0$:

$$x = \frac{k}{s^2 + 2\zeta\omega_n s + \omega_n^2} u \qquad \rightarrow \qquad x = \frac{b_0}{a_0 s^2 + a_1 s + a_2} u$$

Similarly, the vehicle speed model was given by a 1st order equation with $n = 1$ and $m = 0$,

$$x = \frac{k}{\tau s + 1} u \qquad \rightarrow \qquad x = \frac{b_0}{a_0 s + a_1} u$$

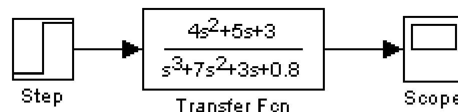
Finally, note that sometimes the coefficients $a_0 \ldots a_n$ and $b_0 \ldots b_m$ are either zero or unity. If you are not sure about all this, hopefully the examples below will help clarify matters!

Exercise 15

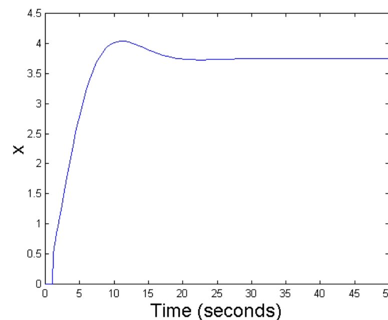
Find the unit step response of the following 3rd order transfer function model.

$$x = \frac{4s^2 + 5s + 3}{s^3 + 7s^2 + 3s + 0.8} u$$

Hint: create the block diagram shown below.



To program these parameters into SIMULINK, double click on the Transfer Fnc and change the settings so that numerator is `[4, 5, 3]` and the denominator is `[1, 7, 3, 0.8]`. Change the simulation length to 50 seconds to see the entire response, as shown below:



Exercise 16

Find the unit Step response of the three models below. In each case, make a brief note of the stability and general form of the output *looking out for any unusual features*. Also, find the poles and zeros of the system and relate these to the response. Find the steady gain of each system by setting $s = 0$ and compare this with the observed response. Finally, examine how each model responds to other types of input signal, such as an Impulse, Ramp or Sine Wave.

$$x = \frac{6s}{s^4 + 2.9s^3 + 5.43s^2 + 7.925s + 4.395} u$$

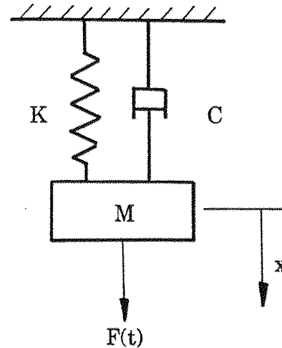
$$x = \frac{5}{s^3 + 0.75s + 0.125} u$$

$$x = \frac{s^2 - 11s + 3}{s^3 + 1.5s^2 + 0.75s + 0.125} u$$

Hints: the numerator polynomial for the 1st example above should be entered as `[6, 0]`. If you just use [6], then SIMULINK will assume a scalar 6 rather than $6s + 0$ as it should be. The numerator for the 2nd example is indeed just [5]. However, make sure that you include a naught for the missing s^2 term in the 2nd example.

4.3 Differential Equations solved using Integrators

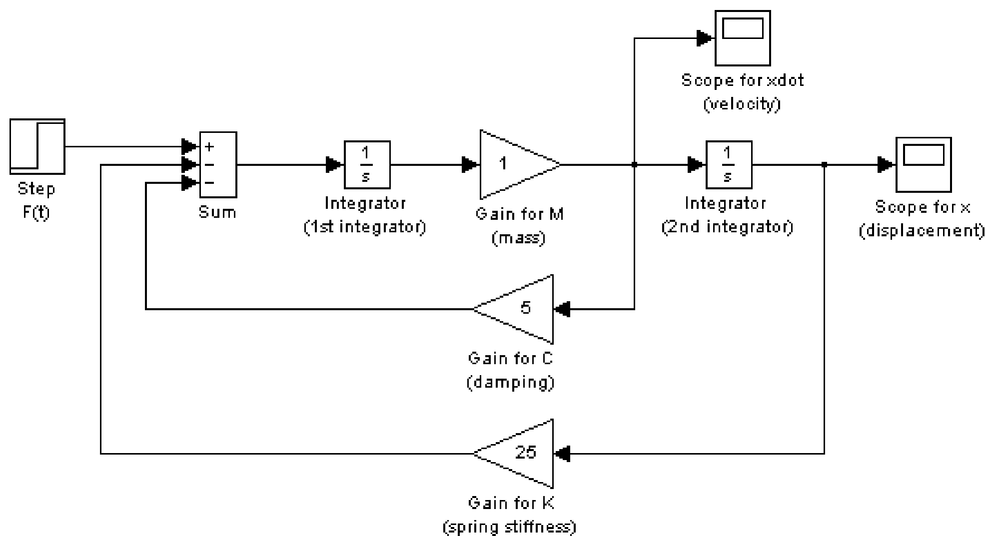
Consider again the mass-spring-damper system shown below, where M is the mass, K is the spring stiffness, C is a parameter for the damping, $F(t)$ is an external force and x is the displacement of the body (measured in metres from an arbitrary reference point). Note that the velocity (the speed at which the body moves away from the reference point) is the rate of change of displacement, denoted by $\dot{x}$.



In contrast to the approach taken in Part 3, one way to mathematically describe the behaviour of the mass is to consider the momentum $p = M\dot{x}$, i.e. momentum equals the mass multiplied by the velocity. Rearranging yields $\dot{x} = p/M$. Using Newton's Law, the mass-spring-damper can then be described by the following pair of 1st order differential equations:

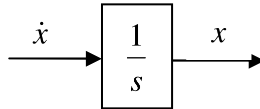
$$\dot{x} = \frac{p}{M}$$
$$\dot{p} = F(t) - Kx - C\dot{x}$$

The equation pair above is represented in SIMULINK using the following block diagram:



Why the above diagram? It might take a while to work it out, but keep comparing the equation pair with the block diagram until you see it! Here's my attempt to explain in words:

Consider an individual `Integrator` block first. SIMULINK takes the input to an Integrator and (unsurprisingly) integrates it, as shown in the figure below. Note that the input signal below is denoted by $\dot{x}$, hence the integrated output is simply x.



Now return to the big SIMULINK diagram on the previous page. Start on the right hand side with the Scope labelled "Scope for x". This is the displacement x. Work your way back towards the left, past the block labelled "2nd Integrator" to find the velocity $\dot{x}$. This velocity is extracted from the block diagram (for plotting) using the Scope labelled "Scope for xdot".

Next consider the summation block labelled simply "Sum". This is a critical part of the diagram. In fact, the output from the summation block is $\dot{p}$ from the 2nd equation:

$$\dot{p} = F(t) - Kx - C\dot{x}$$

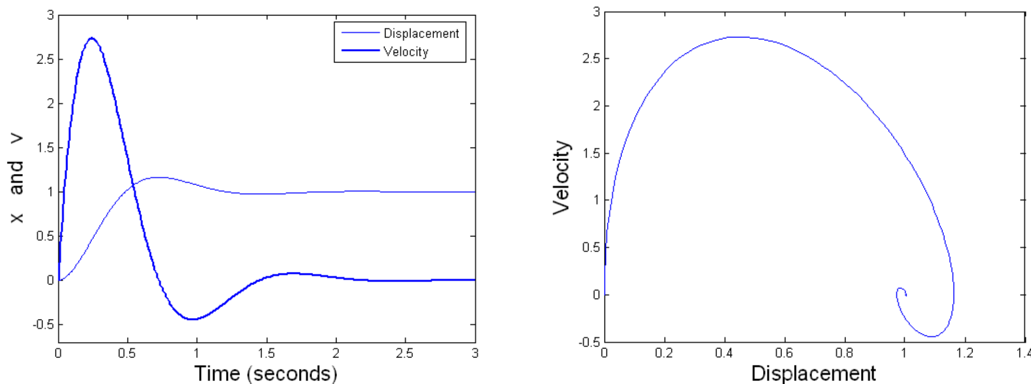
Note that the "Step F(t)" is an input to the summation block. Subtracted from the summation is xdot (or rather $\dot{x}$) multiplied by a gain for C (here given an arbitrary value of 5 units). Also subtracted is x multiplied by a gain for K (here given an arbitrary value of 25 units). Finally, the integrator labelled "1st Integrator" takes $\dot{p}$ from the summation and integrates it, with the output multiplied by the Gain for $1/M$ (here the mass is 1kg) to represent the first equation:

$$\dot{x} = \frac{p}{M}$$

As you can see, it is rather awkward to explain all this in words, so the best approach is to think about it, discuss with your neighbour in the lab and/or ask the tutor or demonstrator.

Exercise 17

Set the simulation length to 3 seconds and graph the response of the mass-spring-damper system to a step change in the force, as shown below. Note that the right hand side graph shows the velocity plotted against the displacement. It is easiest to prepare these graphs using a script called by MATLAB, as shown in earlier laboratories.



The advantage of using a system of 1st order differential equations (rather than a Transfer Function), is that any intermediate variables are available for plotting. For example, here both displacement and velocity can be examined, whilst in Part 3 Exercise 12 the Transfer Function only provided the displacement variable.

Another advantage is that nonlinear and other "non-standard" models can be represented using a combination of Integrators, as illustrated by the exercises below:

Exercise 18

Some of these examples do not include an input variable. In this case, to see a non-zero response it is necessary to specify initial conditions for the Integrators, as mentioned in the relevant problem. You might have to think about some of these, but have a go before checking the solution!

Problem 18.1

Obtain the unit step response of the system below using two different approaches: (i) a *Transfer Fnc* block and (ii) an *Integrator* block. Check that the response is the same.

$$\dot{x} = -x + u$$

Problem 18.2

If a vibratory system is subject to a vibratory force with a frequency that is slightly different to the natural frequency, then the system exhibits the beat phenomenon. Obtain the response of the system: $\ddot{y} + \omega^2 y = \sin(t)$ when $\omega^2 = 1.1$ and $\omega^2 = 0.8$. What happens if $\omega^2 = 1$?

Problem 18.3

Obtain the response of the system: $\dot{x} = -x + \sin(10x) + \sin(t)$

Use the *Fcn* block of the *Functions & Tables* group. Double click on this block and type `sin(10*u)` in the parameter field to obtain $\sin(10x)$.

Problem 18.4

Obtain the response of the system: $\dot{x} = -x + x^2$

with the initial condition $x(0) = 0.2$. Double click on the *Integrator* block and type the initial condition in the parameter field. The quadratic term may be set up by means of the *Product* block of the *Math* group. Then consider the response for the initial conditions 0.5, 1.0, and 1.5. Note that the system is stable if $|x(0)| < 1$.

Problem 18.5

Obtain the unit step response of the system: $\ddot{x} + 0.5\dot{x} + 2x + x^2 = u$

Note that this problem cannot be solved using a Transfer Function because of the x^2 term.

Problem 18.6

Study in the phase plane (x on the horizontal axis and $\dot{x}$ on the vertical axis) the behaviour of:

$$\ddot{x} + \dot{x} = 0$$

for a variety of initial conditions.

Problem 18.7

Study the behaviour of the system:

$$\ddot{x} + \left|x^2 - 1\right|\dot{x}^3 + x = \sin\left(\frac{\pi x}{2}\right)$$

for a variety of initial conditions.

4.4 Case Studies

The following research articles use SIMULINK models to study practical control problems:

- Gu, J., Taylor J. and Seward, D., (2004), Proportional-Integral-Plus control of an intelligent excavator, *Journal of Computer-Aided Civil and Infrastructure Engineering*, **19**, 16-27.
- Taylor, C.J., Mckenna, P.G., Young, P.C., Chotai, A. and Mackinnon, M., (2004), Macroscopic traffic flow modelling and ramp metering control using Matlab/Simulink, *Environmental Modelling and Software*, **19**, 10, 975-988.

The following describes a MATLAB toolbox for time series analysis, forecasting and control:

- Taylor, C.J., Pedregal, D.J., Young, P.C. and Tych, W., (2007), Environmental Time Series Analysis and Forecasting with the Captain Toolbox, *Environmental Modelling and Software*, **22**, 6, 797-814.

A trial version of the toolbox mentioned above can be downloaded from:

- http://www.es.lancs.ac.uk/cres/captain/