L.1: Introduction to MATLAB

L.1.1 Learning Objectives

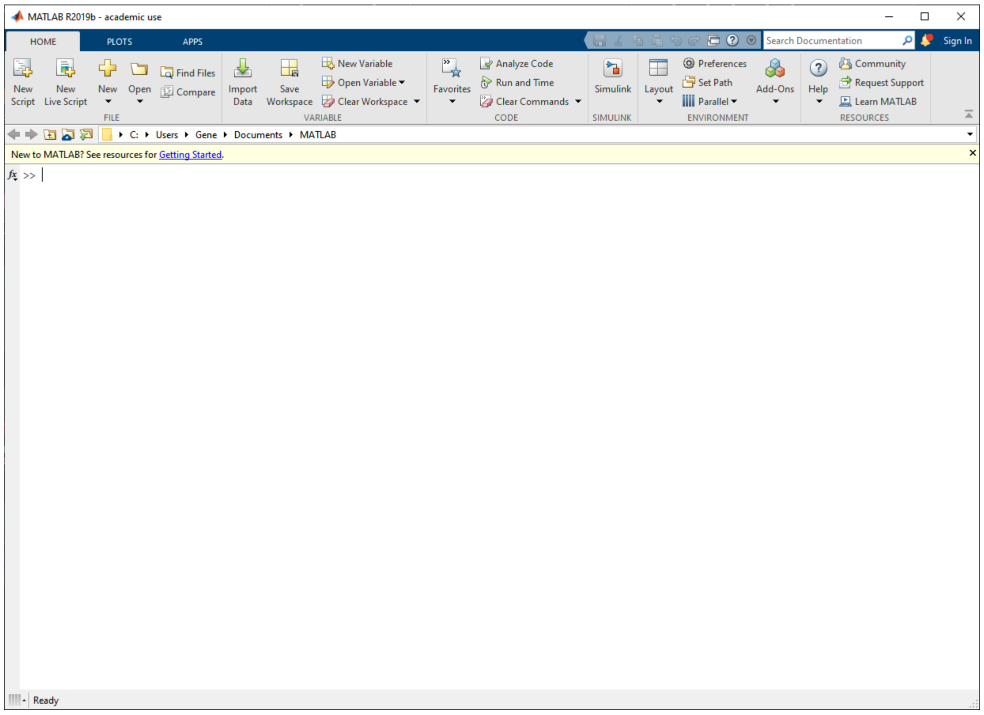
This appendix provides a basic introduction to MATLAB. By following through it as a step-by-step tutorial, you will learn to:

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

Before leaving the lab, make sure you have accomplished these objectives – these skills will be needed later on!

L.1.2 The 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 we will see n this section. 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 look somewhat differently from that shown below. To avoid later confusion in these notes, do the following before continuing: From the **Environment** section of the ribbon menu at the top of the MATLAB window, select **Layout > Command Window Only.** The MATLAB window should now look something like this:


MATLAB R2019b - academic use

New to MATLAB? See resources for Getting Started.

```
>>
```

Type `1+2` and press the Enter key. The result is assigned to the generic variable `ans` and is printed on the screen.

```
>> 1 + 2

ans =

     3
```

Not too difficult so far! You can quit MATLAB at any time by typing `quit` in the command window.

L.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 enter after typing a command:

```
>> 100 / 5

ans =

    20

>> 100 / 5 / 2

ans =

    10

>> 100 + 5 / 2

ans =
  102.5000

>> (100 + 5) / 2

ans =

   52.5000

>> 100 + 5 ^ 2 / 2

ans =

  112.5000

>> 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

a =

    10

>> b = 100

b =

   100

>> a * b

ans =

        1000
```

Variables can be reassigned:

```
>> a = 10

a =
     10

>> b = 100

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`.

```
>> who

Your variables are:

a    ans  b
```

Type the name of one of these variables and press return to see its current value:

```
>> a

a =

        1000

>> b

b =

   100
```

Exercise L.1.1

Use MATLAB to evaluate the following:

(i) $100+5^3$

(ii) $\dfrac{100+5}{2+10}$

(iii) $3\times100+\frac{97\times5}{2+20}$

L.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 intended to sum two variables and then divide the result by 2, but we mistakenly left out the closing right parenthesis.

```
>> b = 100

b =

   100

>> c = 5

c =

     5

>> (b+c/2)
  (b+c/2
         ↑
Error: Invalid expression. When calling a function or indexing a
variable, use parentheses. Otherwise, check for mismatched
delimiters.

Did you mean:
>> (b+c/2)
```

Note that, even though it's wrong this time, MATLAB also suggests a possible correction for the error. If the suggestion is what you intended, then just press Enter to use it.

Also, at any time, you can use the cursor keys to:

- Use **left** / **right** arrow keys to edit text on the command line,
- Use the **up** / **down** arrow keys to scroll back to find previously entered commands, and use them again.

Exercise L.1.2

Use the arrow keys to edit and correct the previous command and find $(b+c)/2$. Check that the answer makes sense.

L.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 elements of 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 notes, the semi-colon is useful when analyzing 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 parentheses. Thus, continuing the previous example:

```
>> a = x(2)

a =

     9

>> x(1)

ans =

   -1.3000

>> -2*x(1)

ans =

    2.6000

>> 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.

L.1.2.4 Elementary functions

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

`sin(X)`	sine of the elements of X in radians,	`sind(X)` for degrees
`asin(X)`	arcsine of the elements of X in radians,	`asind(X)` for degrees
`cos(X)`	cosine of the elements of X in radians,	`cosd(X)` for degrees
`acos(X)`	arccosine of the elements of X in radians,	`acosd(X)` for degrees
`tan(X)`	tangent of the elements of X in radians,	`tand(X)` for degrees
`atan(X)`	arctangent of the elements of X in radians,	`atand(X)` for degrees
`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)`	base 10 logarithm of the elements of X	
`exp(X)`	exponential of the elements of X	
`sum(X)`	sum of the elements of X	

`length(X)`	length 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
```

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 =

    1.0000    0.5403
```

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

```
>> clear A
>> who

Your variables are:

B     a     ans   b     c     x
```

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

ans =

    3.1416

>> pi = 5

pi =

     5
```

Oops, this is probably a bad idea! To correct it, type the following:

```
>> clear pi
>> pi

ans =

    3.1416
```

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

```
>> 0 / 0

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 variables named 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 =

   3.141592653589793

>> format short e
>> pi

ans =

   3.1416e+00

>> format long e
>> pi

ans =

     3.141592653589793e+00
```

Finally, return to the standard format:

```
>> format short
>> pi

ans =

    3.1416
```

L.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 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 set of *x* and *y* values 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 element-by-element operations involve multiplication, division or powers, the operator must be preceded by a dot. Further examples are 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.)

L.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  *
Incorrect dimensions for matrix multiplication. Check that the
number of columns in the first matrix matches the number of rows
in the second matrix. To perform elementwise multiplication, use
'.*'.
```

However, a row vector can be multiplied by a column vector as follows. First of all, take the transpose of `B` using the apostrophe 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 using a semi-colon to start each 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 L.1.3

1. Experiment with the functions `ones`, `zeros` and `eye` to learn how they work. Use these functions to create the following:
    a. a 5 x 5 diagonal matrix with its diagonal elements all equal to 8
    b. a 6 x 6 matrix with all of its elements equal to 3.5
2. Evaluate the following expressions, with: A = 100, B =5, C = 2, D = 10.

    a. $\frac{A+B+C}{2D}$

    b. $A+B^C$

    c. $\frac{-1}{B(A-B)}$

    d. $\frac{AD}{BC}$

3. Calculate the areas of circles of radius 1, 1.5, 2, 2.5, ..., 10m. Hint: to quickly solve all nineteen cases at once, first define an array for the radius, e.g. `rad = [1:0.5:10]`.
4. Calculate the areas of the rectangles whose lengths and widths 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
width	1	5	2	0.5

L.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

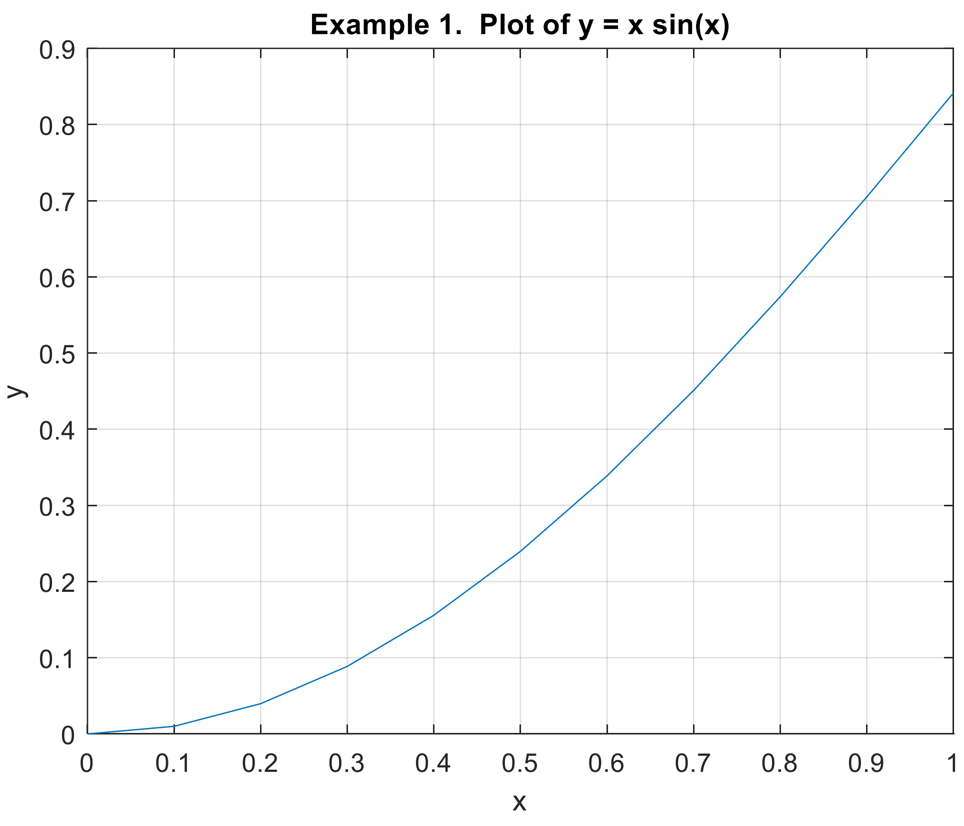
The axis scales and line types are automatically chosen. However, graphs may be customized using the following functions:

`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 coordinates
`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.  Plot of y = x sin(x)')
>> xlabel('x')
>> ylabel('y')
>> grid on
```

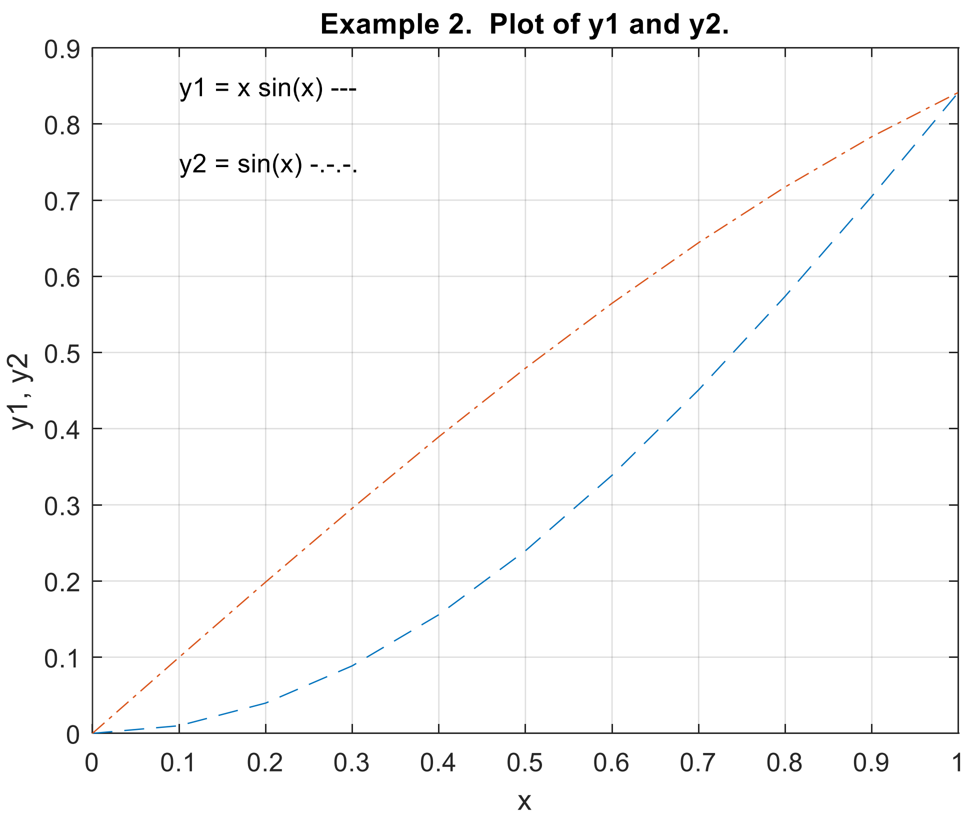


Plots may include more than one line and line types may be specified in the plot statement. A few examples are:

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

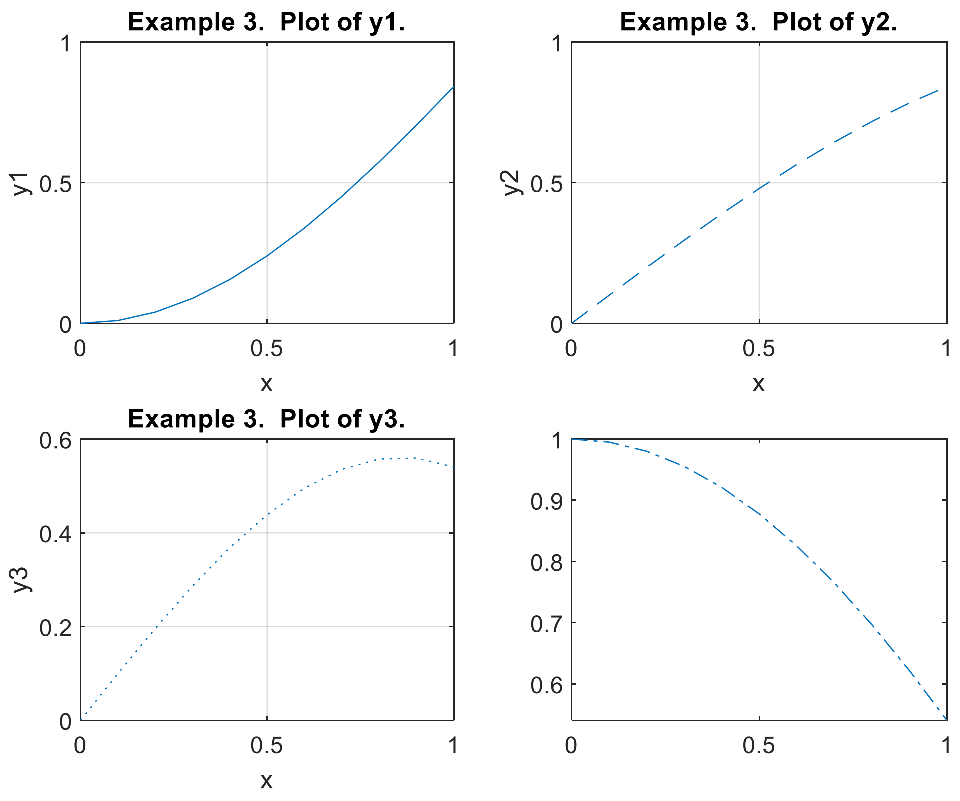
Type `help plot` to see a full list of all the color and line type options. Normally, the `plot` command clears any previous lines in the same figure window. However, `hold on` freezes the figure and is useful for graphing multiple lines, as shown in the next example. Here, for brevity, some of the commands are written on the same line, separated by a semicolon – you can either type the example this way, or write the commands 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,'-.')
>> grid on
>> title('Example 2.  Plot of y1 and y2.')
>> xlabel('x'); ylabel('y1, y2')
>> text(0.1,.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 that are inexplicably numbered consecutively. This facility is illustrated in the next example.

```
>> 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,'-')
>> grid on
>> title('Example 3.  Plot of y1.')
>> xlabel('x'); ylabel('y1')
>> subplot(2,2,2); plot(x,y2,'--')
>> grid on
>> title('Example 3.  Plot of y2.')
>> xlabel('x'); ylabel('y2')
>> subplot(2,2,3); plot(x,y3,':')
>> grid on
>> title('Example 3.  Plot of y3.')
>> xlabel('x'); ylabel('y3')
>> subplot(2,2,4); plot(x,y4,'-.')
>> grid on
>> title('Example 4.  Plot of y4.')
>> xlabel('x'); ylabel('y4')
>> subplot(2,2,4); plot(x,y4,'-.')
```



Point of Interest: This document was created using Microsoft Word. The MATLAB graphs were copied to the clipboard using the MATLAB Figure Window Menu: **Edit > Copy Figure**, and then pasted into the Word document.

Exercise L.1.4

Fahrenheit and Celsius temperatures are related by:

$$F = \frac{9}{5} \times C + 32$$

Plot a 2×1 array of graphs, with:

a. Fahrenheit vs. Celsius over the range $-50 \le C \le 150$ in the upper window, and
b. The inverse relationship over the range $-50 \le F \le 250$ in the lower window.

L.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 Appendix L.2) 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). To do this, one may create a script called `plotsine.m`, as shown below. You can create it by using the MATLAB Editor: select the New Script item on the MATLAB File Menu, which is to be found at 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! This also 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 you must change to this directory using the `cd` command, as follows, before continuing:

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

Next, type in a value for omega:

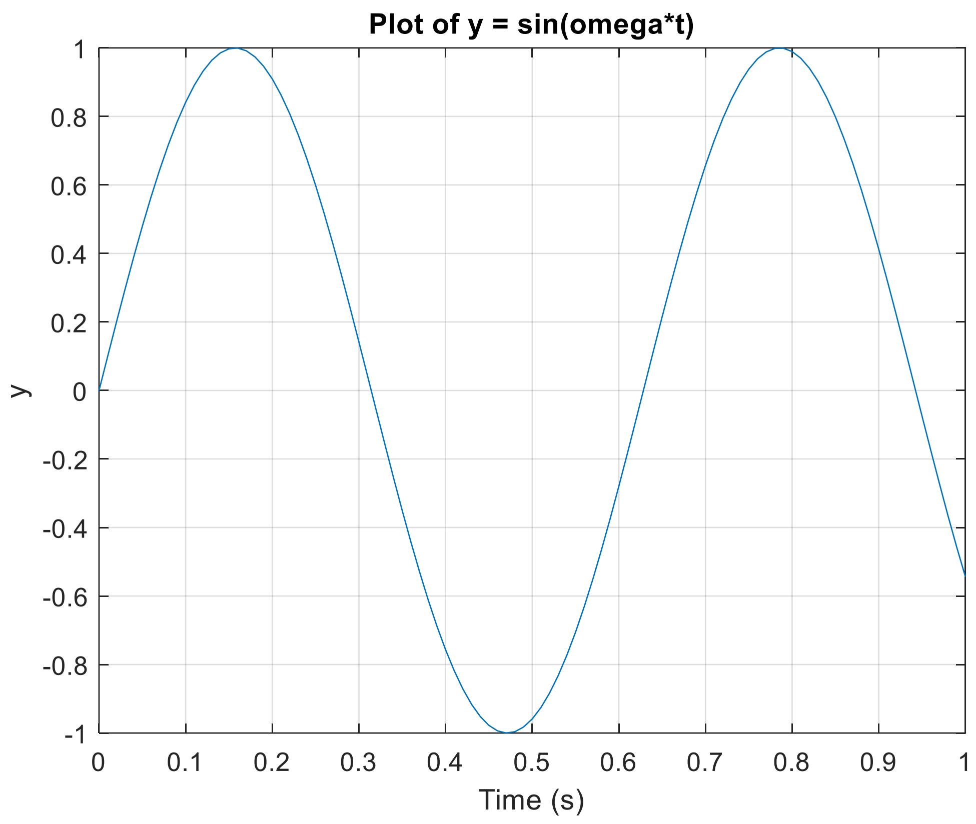
```
>> omega=10

omega =

   10
```

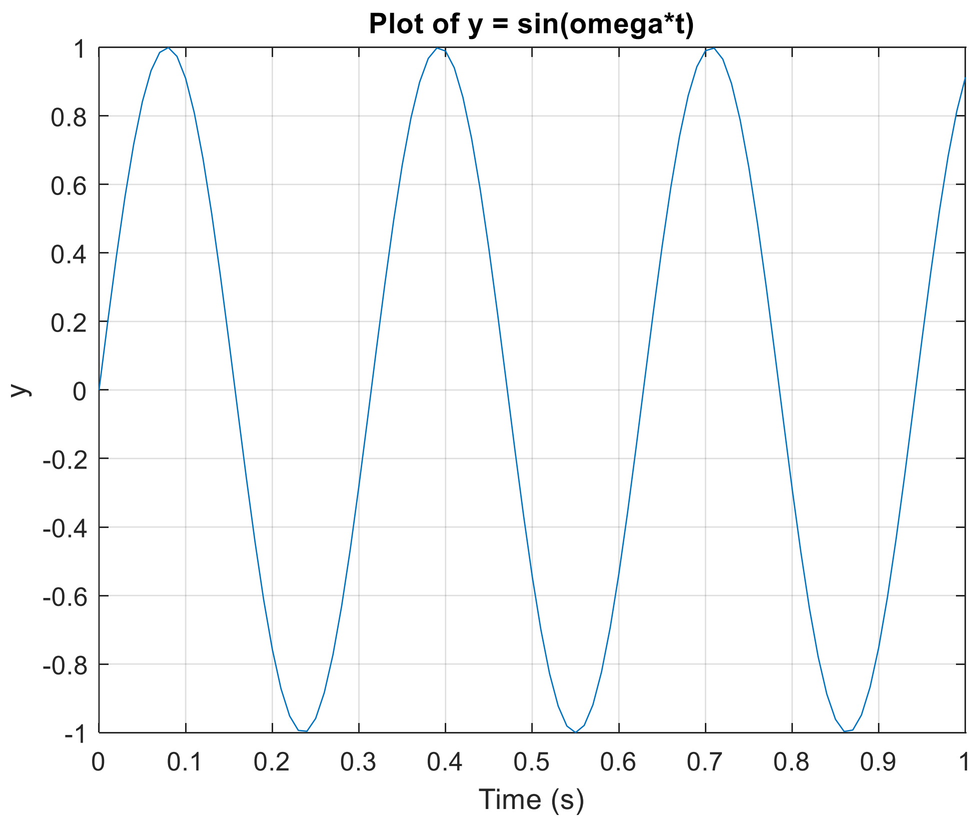
Finally enter the name of your script in the command window and press the Enter key.

```
>> plotsine
```



If you wish to change the value of omega, you should 're-run' the script to update the plot, as shown below:

```
>> 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]; width = [1 5 2 0.5];
area = length.*width;
disp('     length     width       area')
disp('       m          m       sq m')
disp([length' width' area'])
```

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

```
>> example
    length     width      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`, `width` and `area` have been printed out as columns rather than rows. The apostrophe after the name changes the array from a row to a column, i.e., vector transpose (see Section L.1.2.6 above). In fact, the expression `[length' breadth' area']` creates a matrix comprising these three columns.

L.1.3.1 For Loops

A `for` loop allows a statement, or group of statements, to be repeated a 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 three statements have been written on one line, terminated by a semicolon to suppress 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.

L.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. Store this as a 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 Windows Notepad or the MATLAB Editor to create the following file called `test.dat` and save it to the PC hard drive or your own flash 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.dat
>> who

Your variables are:

i       m       n       test    x       yclosed  yopen
```

Note that the file extension `.dat` is required to load the data, but by default MATLAB then assigns the 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.dat
```

L.1.3.4 Matrix Elements

The test variable 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:

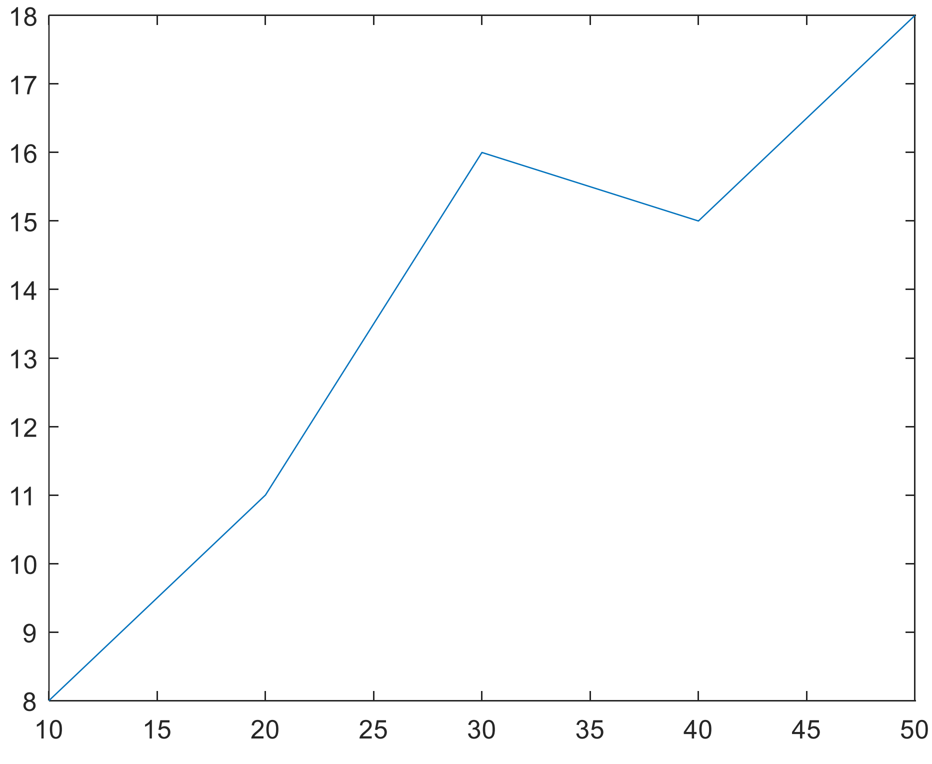
```
>> test(3,2)
ans =
    16
```

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:

```
>> test(3,:)
ans =
    30    16
>> test(:,2)
ans =
     8
    11
    16
    15
    18
```

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

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

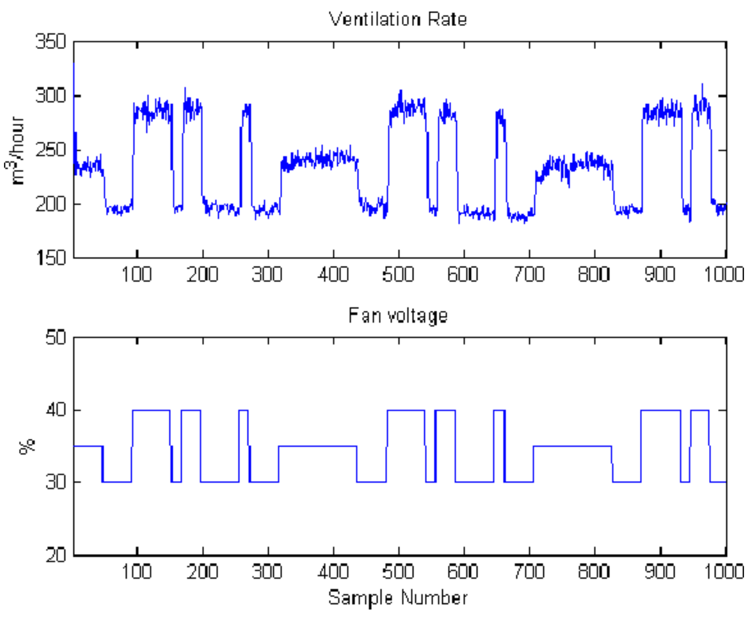


Exercise L.1.5

For this exercise, three sample data files are provided through the class website at http://www.stuffle.website. Each file represents one particular experiment in a study of control of the ventilation rate in an instrumented micro-climate test chamber. The data show how the ventilation rate responds to a given voltage test signal. Download the three files (`Data_File_1.dat`, `Data_File_2.dat` and `Data_File_3.dat`) to your working folder or another storage location in your MATLAB path.

The first column of each data file is the ventilation rate (m^3/hour), and the second column is the input fan voltage expressed as a percentage. The data are sampled every 2 seconds (one sample in each row of the file).

1. For `Data_File_1.dat`, write a script to load the data into MATLAB and generate a pair of graphs similar to the figure shown below. (Note that the variable on the abscissa is the sample number in each case.) You will need to use the commands: `load`, `subplot`, `plot`, `title`, `ylabel`, `xlabel` and `axis` to do this. Use the `help` command for more information about this new `axis` function. Can you figure out how to use it?



2. Use your script to very quickly plot the data from the other two files. If you have written an appropriate script for the first task 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 had typed out the commands out one by one, or used the menu system to annotate the graph, then it would be much more work!