University Of Nottingham

Introduction to MATLAB for Engineers

Louise Brown, John Crowe, David Hann, David Morris

April 2019

Contents

1 Introduction to the MATLAB desktop

After starting MATLAB for the first time the desktop will be displayed. The default layout is shown in Fig 1.

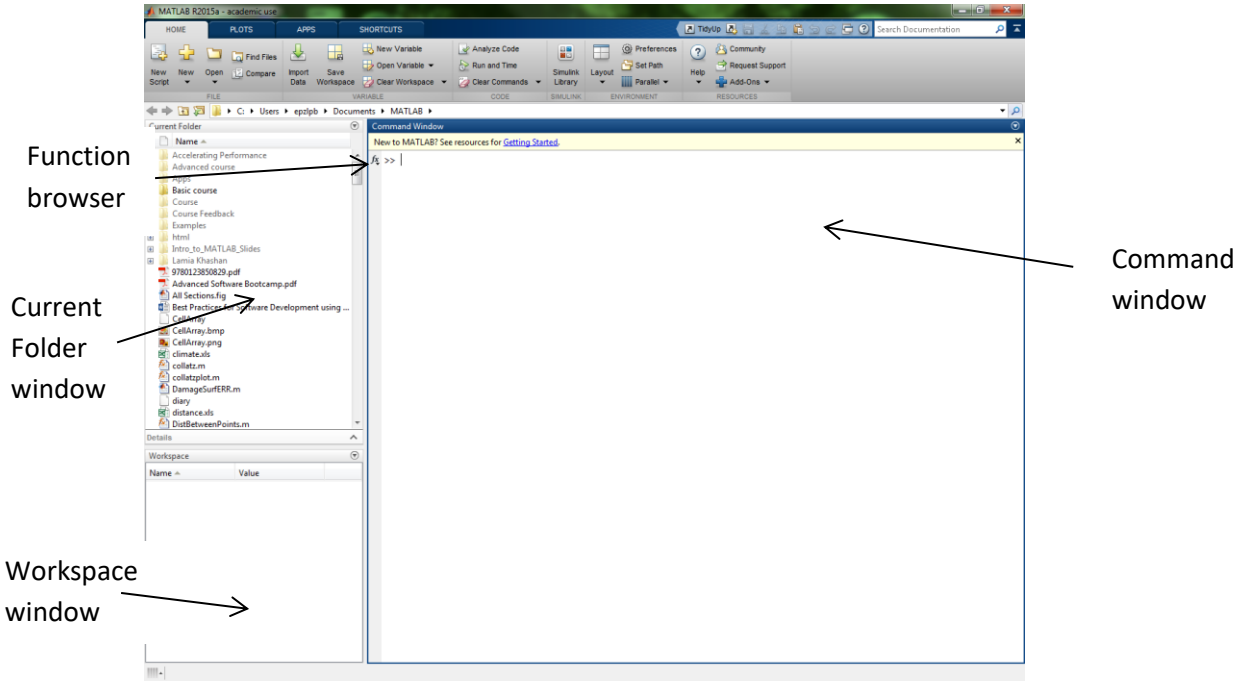

Figure 1. Default MATLAB desktop environment

The largest window is the Command Window. This can be used as an interactive scratchpad, like a calculator, by simply typing commands or expressions at the >> prompt. Maths and other expressions are included in the form of functions and a list of these can be obtained by selecting the f_x symbol next to the >> prompt.

To the left of the Command Window is the Current Folder window which shows all files in the current working folder. A drop-down menu and folder options above the Command window show the path for this folder and allow it to be changed. Any files loaded or saved will, by default, be to or from this folder. The path command will show a list of files in the current path. This is the order in which MATLAB will search for functions. Using the command `path( path, 'mypath')` allows the new folder 'mypath' to be added to the path. This can be useful if developing a set of programs in a folder which can be made to be generally accessible to all MATLAB programs, irrespective of the current working folder. Other `path` functions can be found in `doc path`.

Also to the left of the Command Window is the Workspace window which shows all of the variables which are currently in memory for the MATLAB session.

Another useful window to open is the Command History window which shows all commands entered in both the current and previous sessions. Selecting Layout->Command History->Docked will open this window.

1.1 Changing the default configuration

The layout of the working environment is a personal thing and can easily be changed to suit the user's preferences. This can be done using options in the Layout menu or by using the options given by the button at the corner of each window. The arrow icon will undock the window allowing it to be moved out of the desktop (useful if you have more than one screen). Windows can be minimised using the sideways pointing arrow. This results in a tab at the side of the desktop which can be used to restore the window. Windows can be stacked on top of each other by dragging the title bar over the title bar of another. The different windows are then accessed by a set of tabs.

The setup can then be saved using Layout -> Save Layout.

Layout->Default will restore the default layout.

Typing `clc` at the command prompt will clear the command window (but not the command history).

`clear` clears all current variables or data from the workspace.

`clear x` clears a specific variable from the workspace (in this case `x`).

1.2 Help

Documentation for functions can be accessed by using either the `help` or `doc` commands entered at the command prompt. `help` plus the function name will give a description of the function in the command window. `doc` gives the full documentation in a separate window. Using an internet search engine is also useful and may provide examples of usage not given in the MATLAB documentation.

2 MATLAB Basics

2.1 Variables

A MATLAB variable is a region of memory known by a specific user name. In MATLAB all variables are stored as arrays. A scalar is a 1x1 array, a vector is a 1xn or nx1 array (a row or column vector) and a matrix is an m x n array.

Variables are created using the assignment statement: `variable = expression`. The expression is evaluated and then assigned to the variable. For example typing the following (where *<ret>* indicates pressing the return key)

```
>> a = 2+3 <ret>
```

Produces

```
a
     = 5
```

The Workspace window now shows a variable called `a` with a value of 5. Double clicking on the variable in the Workspace Window will open a Variable Editor window which shows that `a` is a <1x1 double>. A grid is also displayed which allows the variable to be changed by typing in different values.

Typing `who` at the command prompt will give a list of variable names.

`whos` will give information including the array (or matrix) size, the number of bytes it occupies and its class, in this case a double.

Typing the variable name at the command prompt will print the contents of the variable to the screen.

Unlike some programming languages the type of the variable does not need to be specified. It will default to a double. Character variables can be created by using single quotes, eg `b ='b'`. Character strings are created with a sequence of characters in single quotes, eg `string ='abcde'`.

Note that there are no increment and decrement operators (++ or --) in MATLAB. Also note that, unlike some other programming languages, multiple assignments are invalid in MATLAB, eg. a = b = 1 is not a valid statement.

Adding a ';' at the end of a statement will supress output to the screen. For longer programs this is useful as outputting to the screen slows down running of programs and its use is the norm for MATLAB programs. (Note that a simple debugging tool can be to temporarily remove the ';' to see the value of variables at a particular point in the program).

Long lines of code may be separated using an ellipsis … at the end of a line. (This is simply typed in as three full stops). The continuation on the following line is then treated as part of the same command or expression.

Variables may be displayed in long or short format using the format statement. For example:

```
>> format long
>> pi
ans =
   3.141592653589793
>> format short
>> pi
ans =
    3.1416
```

Note that this simply affects how the variables are displayed on screen and not how they are saved in memory. Type `help format` for other format options.

2.1.1 Variable names

These must start with a letter but may include _ and numbers.

Use meaningful names, it makes code much easier to read.

They are case sensitive.

Certain reserved words cannot be used as variable names. Type `iskeyword` at the prompt to find out what these are.

Beware of using built in function names as variable names, eg MATLAB will allow you to call a variable sin or cos but then you will no longer have access to these functions. (If you do inadvertently do this clearing the workspace will restore them to their default functions). Similarly beware of overwriting `pi`, `i` and `j`. `i` and `j` can safely be used as iterators if you are not using complex numbers. Otherwise their default meaning is $\sqrt{-1}$. Typing `i` or `j` at the command prompt will display the value as a complex number, `0 + 1.0000i`. Note that taking the square root of a negative number in MATLAB will not result in an error but give a complex number as the result, for example:

```
>> sqrt(-2)
ans =
        0 + 1.4142i
```

2.2 Vectors

A vector (ie a one dimensional array) can be created in several ways:

```
>> A = [1 2 3 4]  or
>> A = [1,2,3,4]
```

both give

```
A =
     1  2  3  4
```

a 1 x 4 row vector. The use of commas or spaces between the vector elements is up to personal preference.
The colon operator can be used to produce regularly spaced elements in a vector.

```
>> A = [1:4]
```

gives the same result as the previous methods, the colon producing regularly spaced numbers from 1 to 4 with a an integer spacing of 1 between the elements. This syntax is used regularly in MATLAB.

The step between the elements can be altered by specifying an extra parameter using the format [first : step : last]. Note that the step does not have to be an integer.

```
>> C = [1:2:9]
```

gives

```
C =
     1  3  5  7  9
```

To create a column vector separate the elements using a ;

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

gives

```
B  =
     1
     2
     3
```

There is also a transpose operator ’

```
>> B = [1:3]’
```

also gives

```
B  =
     1
     2
     3
```

Linearly spaced arrays can also be created using the `linspace` function. This will produce a vector with a given number of equispaced elements from x to y in n steps using the format `linspace( x, y, n )`. Note that this is a function call and so there must be a comma between the parameters. The spacing between the elements is automatically calculated by the function.

For example

```
>> vec = linspace( 1, 8, 5 )

vec =

     1.0000    2.7500    4.5000    6.2500    8.0000
```

Logarithmically spaced arrays can be created using the `logspace` function. This generates a vector n points between decades 10^a and 10^b using the format `logspace(a,b,n)` as shown below:

```
>> vec = logspace(1,2,5)

vec =

   10.0000   17.7828   31.6228   56.2341   100.0000
```

There are numerous functions within MATLAB which perform array operations. The MATLAB->Language Fundamentals->Matrices and Arrays->Array Dimensions section of the Function Browser shows basic operations such as `size()` and `length()`. Functions for manipulating arrays can be found in the Matrices and Arrays->Sorting and Reshaping Arrays section. Basic statistical functions such as mean, mean(), and variance, var(), can be found in the MATLAB->Mathematics->Statistics and Random Numbers->Descriptive Statistics section.

2.2.1 Indexing Vectors

Individual elements of an array are referred to using subscript notation (note that indices start from 1, not 0 as in some other programming languages). In the following example `a(4)` extracts the 4th element of the vector.

```
>> a = [1:2:14]
a =
     1     3     5     7     9    11    13

>> b = a(4)
b =
     7
```

The subscript can also be another vector enabling the extraction of a set of items from the vector. The example below creates a vector which is then immediately used to index into the vector `a`. This could also be a predefined vector, as shown in creating `d` below.

```
>> c = a( [1 4 5] )
c =
     1     7     9
```

```
>> index = [1 3 6]
index =
     1     3     6
>> d = a( index )
d =
     1     5    11
```

Colon notation can be used to extract a range of elements from a vector. The example below extracts the 3rd to 6th elements.

```
>> a(3:6)
ans =
     5     7     9    11
```

The last element can be referred to using the term `end`. This can be used either on its own or as part of a range.

Extract the last element:

```
>> a(end)
ans =
    13
```

Extract the 4th to last elements:

```
>> a(4:end)
ans =
     7     9    11    13
```

The order of the elements can be reversed using

```
>> a(end:-1:1)
ans =
    13    11     9     7     5     3     1
```

Note that this will initially create a vector from the last to first index using steps of -1 and then use this to index into the vector.

Certain elements of a vector can be replaced by using an indexing expression on the left hand side of the expression. The values on the right hand side will be assigned to the elements selected on the left. In this case the 1st, 3rd and 5th elements of the vector are replaced.

```
>> a( [1 3 5] ) = [20 30 40]
a =
    20     3    30     7    40    11    13
```

The following will replace the selected elements of `a` with the scalar value 40.

```
>> a( [1 3 5] ) = 40
a =
    40     3    40     7    40    11    13
```

Exercise 2.1

1) Create a row vector A with odd numbers from 5 to17
2) Create a column vector B with decreasing numbers from 3 to -3
3) Create a vector C containing the 1st, 4th and 6th elements of vector A
4) Replace the 3rd, 5th and 7th elements of vector B with vector C

2.3 Operators

Arithmetic operations in MATLAB are as follows:

Addition	`a + b`
Subtraction	`a – b`
Multiplication	`a * b` for matrix or scalar multiplication
	`a.*b` for element by element multiplication
Division	`a / b` for matrix or scalar division
	`a./b` for element by element division
Left division	`a \ b` for matrix or scalar left division
	`a.\b` for element by element left division
Exponentiation	`a ^ b` for matrix or scalar exponentiation
	`a.^b` for element by element exponentiation

The order of precedence for arithmetic calculations is: Evaluate parentheses working from the inside to the outside, exponential, * and / working from left to right, + and – working from left to right.

It is important to appreciate how MATLAB uses array operators. Where in most programming languages a loop may be required to perform an operation on every element of a vector or array, in MATLAB the same can be achieved using a single operation. This is more efficient both in terms of writing code and in making use of the built in optimisation for array operations in MATLAB. This is often called vectorisation.

Scalar operations can be achieved simply, multiplying or dividing each member by a scalar, or adding or subtracting a scalar to or from each member. For example to multiply each element of the vector by 7:

```
>> A = [1:5]
A =
     1  2  3  4  5
```

```
>> A = A * 7
A =
     7 14 21 28 35
```

And then add 2 to each element:

```
>>A = A + 2
A =
   9 16 23 30 37
```

Array operations such as addition and subtraction will be performed on arrays element by element. So, to add two vectors together:

```
>>A = [1:4]
A =
     1 2 3 4
>> B = [2:5]
B =
     2 3 4 5
>> C = A + B
C =
     3 5 7 9
```

The arrays must have the same dimensions or the operation will result in an error.

Multiplication based operations (*, / , \ , ^) have different meanings when applied to arrays. In order to execute element by element operations in these cases a . must be placed before the symbol.

```
>> D = A.*B
D =
     2 6 12 20
```

The .* operation above multiplies corresponding elements of the two arrays together element by element. `A*B` would result in an error as the two arrays would not have correct dimensions for normal array multiplication.

A legitimate matrix multiplication would be achieved as follows:

```
>> A = [1:4]
A =
     1     2     3     4

>> B = [1:4]'
B =
     1
     2
     3
     4
```

```
>> A*B
ans =
    30
```

Vectors and arrays can be used as input arguments for functions (both built-in and user defined). To find the sin of each member of a vector simply requires

```
result = sin(array)
```

where `result` would be of the same dimensions as array and contain the sin of each individual element of array.

Exercise 2.2

1. Calculate the value of $\frac{1}{2+3^4} + \frac{5}{6*7} + \frac{7}{8}$
2. Create a vector A with 9 equispaced points between 0 and 2pi. Calculate the sine of each angle (using the `sin` function) and store the results in a vector called SinAng.
3. Add 1 to each value in SinAng
4. Create a second vector, B, with the values 1 to 9
5. Multiply together corresponding elements of B and SinAng

2.3.2 Left Division

The left division operator in MATLAB is defined as `a\b = a^-1b` for matrix operations and as `a.\b = b/a` which is performed element-by-element on arrays or when one of the elements is a scalar. The use of the .\ operator is shown here:

```
>> A = [1:4]
A =
     1     2     3     4

>> A.\2
ans =
    2.0000    1.0000    0.6667    0.5000
```

The matrix left operator is especially useful for simple solution of sets of simultaneous linear equations.

The pair of simultaneous equations

$$3x_1 + 2x_2 = 12$$
$$x_1 + 4x_2 = 14$$

can be written as $Ax = B$

where $A = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$, $B = \begin{bmatrix} 12 \\ 14 \end{bmatrix}$, and $x = \begin{bmatrix} x1 \\ x2 \end{bmatrix}$

Solving using linear algebra gives $x = A^{-1}B$

In MATLAB the solution is obtained easily using the left division operator:

```
>> A = [3 2; 1 4]

A =
     3     2
     1     4



>> B = [12;14]
B =
    12
    14

>> x = A\B
x =
     2
     3
```

2.4 Matrices

A two dimensional matrix (or array) is produced in a similar way with spaces or ',' separating row elements and ; starting a new row.

```
>> A = [1 2 3 4 5; 5 6 7 8 9] or >> A = [1:5;5:9]
```

both give

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

Note that in MATLAB matrices must have the same number of elements in each row.

Exercise 2.3

Write the MATLAB code to solve the following set of simultaneous equations:

$$x_1 + 2x_2 + 3x_3 = 13$$
$$2x_1 + 3x_2 - x_3 = 4$$
$$4x_1 - x_2 + 2x_3 = 13$$

2.4.1 Character Arrays

As with numerical arrays, character arrays must have the same number of elements in each row. If creating an array from several strings of different length they must be padded using

spaces to create a valid matrix. An error will be generated if an attempt is made to create a matrix as follows:

```
>> charArray = ['one';'seven';'five']

??? Error using ==> vertcat

CAT arguments dimensions are not consistent.
```

Padding each string with spaces so that they are the same length will allow a valid matrix to be created:

```
>> charArray = ['one  ';'seven';'five ']
charArray =
one
seven
five
```

This can be achieved more easily using the `char` function which will automatically pad the strings:

```
>> char('one','three','seven')
ans =
one
three
seven
```

Since Matlab 2016 an alternative 'string' type has been added which allows the creation of string scalars. In this case a group of characters can be created which are stored as a single string scalar:

```
>> str = "one"
str =
     "one"
```

These strings can then be combined into arrays. Because each string is one item they can be combined into columns without the problems described above for character vectors:

```
>> strArray = ["one"; "seven"; "five"]

strArray =
   3x1 string array
      "one"
      "seven"
      "five"
```

Strings can also be combined using the addition operator

```
>> "one" + " seven" + " five
```

```
ans =

     "one seven five"
```

2.4.2 Matrix indexing

In the same way as for vectors, individual elements of a matrix are referred to using subscript notation (starting from 1), specifying both the row and column of the element:

```
>> A = [1:5;5:9;9:13]
A =
     1     2     3     4     5
     5     6     7     8     9
     9    10    11    12    13

>> A(3,2)  % Extract element in row 3, column 2
ans =
     10

>> A(3,2) = 30  % Change the value of element at (3,2)
A =
     1     2     3     4     5
     5     6     7     8     9
     9    30    11    12    13
```

The row or column subscripts can be vectors.

```
>> A(2:3,3:4)  % Extract rows 2 to 3 and columns 3 to 4
ans =
     7     8
    11    12
```

The colon character can be used to address a whole row or column of a matrix. `A(:,2)` refers to everything in column 2 of A. `A(2,:)` refers to everything in row 2 of A. `The :` here is equivalent to `1:end`.

```
>> A(2,:)  % Extract row 2
ans =
     5    6     7     8     9

>> A(:,[1 3 5]) % Extract columns 1, 3 and 5
ans =
     1     3     5
     5     7     9
     9    11    13
```

Matrices can also be indexed using one subscript whereby the index is obtained by counting down each column in turn until the appropriate element is reached. This is called *linear indexing*.

```
>> A(5)  %  Extract element 5, equivalent to A(2,2)
ans =
    6
>> A(10)  %  Extract element 10, equivalent to A(1,4)
ans =
     4
```

The function `sub2ind(size, row vector, column vector)` is a convenient way of converting from subscript to linear indexing.

```
>> % Obtain linear indices for elements (1,3) and (3,5)

>> index = sub2ind( size(A), [1 3], [3 5] )
index =
     7    15
>> A(index)  % Extract elements using linear index vector
ans =
     3    13
```

The corresponding `ind2sub(size,index vector)` function converts from linear to subscript indexing, returning row and column vectors.

```
>> Obtain subscript indices for elements 7 and 15
>> [row column] = ind2sub(size(A), index)
row =
     1     3
column =
     3     5
```

2.4.3 Further matrix functions

It is also possible to build up larger arrays from smaller ones, eg the matrices A and B could be combined as follows.

```
>> A = [1:5;5:9]
A =
     1     2     3     4     5
     5     6     7     8     9

>> B = [1:2:9;2:2:10]
B =
     1     3     5     7     9
     2     4     6     8    10
```

```
>> D = [A;B]
D =
     1  2  3  4  5
     5  6  7  8  9
     1  3  5  7  9
     2  4  6  8  10

>> E = [A B]
E =
     1  2  3  4  5  1  3  5  7  9
     5  6  7  8  9  2  4  6  8  10
```

The length and size functions can be used to obtain dimensions of an array. `length(B)` will return the size of the largest dimension of an array, in this case 5. `size(B)` will return a matrix containing the length of each dimension of the array. Here it will return `[2 5]`. Note that either may be used and will give the same result for a vector.
The size function can also be used in the form `size(x, dim)` where dim indicates the dimension of the array (1 for rows, 2 for columns and so on for arrays of greater dimensions).

There are a number of matrices which can be generated automatically. The most common of these are the identity matrix, `eye()`, an array of all ones, `ones()`, and an array of all zeros, `zeros()`. A full list can be found in the Function Browser under MATLAB->Language Fundamentals->Matrices and Arrays->Array Creation and Concatenation.

The identity matrix is square and therefore only one dimension is given:

```
>> eye(2)
ans =
     1     0
     0     1
```

For arrays of ones or zeros each dimension must be specified:

```
>> ones(2,4)

ans =

     1     1     1     1
     1     1     1     1
```

2.4.4 Matrix memory management

Extra elements can be added to arrays simply by allocating an element at a given position in the array. If just one element in, for example, a column was added then corresponding empty elements in all of the rows in that column would also be added. It should be noted that each time the size of an array is increased a block of memory will be allocated to fit the new size of the array and the contents moved to that location. If an array is being 'grown' one element at a time this is very inefficient (and potentially slow). If the size of the final array is known then it is preferable to pre-allocate an array of the given size with all of the elements set to zero (or some other convenient value). When the value at a particular location in the array is assigned the value at that particular memory location will simply be reassigned.

Another way of saving memory is to make use of the sparse matrices provided by MATLAB. The simplest way of creating these is to use the `sparse(N)` form of the function which will squeeze out any zero values. The advantage of this can be seen with the example of a 1000 x 1000 identity matrix which will occupy 8MB in full format but only 24008B when stored in sparse format. For full documentation on sparse matrices type `doc sparse`.

Information on sparse matrix functions can be found in the Function Browser under MATLAB->Mathematics->Sparse Matrices.

Exercise 2.4

1) Create the following matrices:

```
A = 9  12  13  0
    10  3   1   5
    2   5  10  3

B = 1   4   2  11
    9   8  16  7
    12  5   0  3
```

2) Use **A** and **B** to create new matrices:

a) **C** – a scalar containing the element in the 3rd row and 3rd column of **A**

b) **D** – a column vector containing the elements in column 3 of **A**

c) **E** – a matrix with 2 rows containing the elements from row 1 of **B** in the first row and row 3 of **B** in the second

d) **F** – a matrix formed by assembling **A** and **B** one above each other

e) **G** – a matrix with 2 columns containing the elements from column 1 of **A** next to those from column 4 of **B**

3) Change some of the entries in these matrices:

a) element (2,2) of **E** to 20

b) change row 1 of **A** to contain all zeros

c) set the values of column 3 of **F** to integer values from 1 to 6

d) set the values of column 1 of **A** to the values in column 2 of **B**

3 Scripts

So far we have simply typed at the command prompt to execute commands immediately. A complete program can be written in a script file to be saved and executed at a later date. In MATLAB these files have a .m extension. A script file can be created by selecting the `New Script` button or by typing `edit` at the command prompt. An edit window will open and the required code can be entered. Alternatively a group of commands can be highlighted in the Command History (select first command then shift/click on the last command), right click and then select Create Script. An edit window will open containing the highlighted code.

Once the required code has been entered it may be executed using Save & Run or, once saved, by typing the script name at the command window. The script will be saved into the Current Folder.

Comment lines start with `%`. Anything on a line after a % will be ignored when the code is run.

As usual, the first line(s) of a script should be a comment (called the H1 line) which will be shown if `help scriptname` is entered to give information about the script.

3.1 Simple input/output

It is often useful to be able to read in values for variables and then to output results at the end of a script.

The simplest form of input is to assign a value typed in by the user in response to a prompt on the screen. This can be done as below:

```
>> value = input( 'Enter a value: ') <ret>
 Enter a value: 10
value =
    10
```

Specifying a second parameter 's' indicates that the input character should be a character or string. If numbers are entered they will be treated as chars. Entering just a space will result in an empty string '' and any spaces included before other characters will be included in the string.

The simplest way to display output is using the `disp` function

```
>> disp( str ) or >> disp('string')
```

would output the value of `str` or the word `'string'` respectively.

A simple way of creating strings to display results can be to combine them into a single vector, for example

```
>> Vel = 20;
>> strTitle = ['Starting speed = ' num2str(Vel) ' m/s']
strTitle =
Starting speed = 20 m/s
```
The num2str function is used to convert the numerical value into a string so that it can be used in the string vector.

4 Saving

Commands entered into the Command Window can be saved using the `diary` or `diary on` function. Any commands will be saved into a file called diary in the Current Folder until the `diary off` command is issued.

Variables can be saved using `save <filename>` or the Save Workspace button. All the current workspace variables will be saved into a file with a .mat extension. Note that this format can only be read into MATLAB but has the advantage that it is independent of operating system as long as MATLAB is running on that system.

To restore the file type `load <filename>` and the variables will be loaded into the workspace.

To save data in ascii format so as to be readable by programs other than MATLAB use `save <filename><variable list> -ascii`. This will save with 8 digit format. Conventionally a .dat file extension is used.

For example

```
>> save('Data.dat','x','y', '-ascii')
```

Exercise 4.1

Write a script called FreeFall.m which calculates the distance travelled by a freely falling object at each time increment between two given times using the equation

$d = 0.5gt^2$

where d = distance travelled by the object
g = acceleration due to gravity
t = elapsed time

The start and end times and time increment will be user input.
Output the distances calculated.
Extra task: Present the output as two columns showing the times and corresponding distances.

5 Plotting

MATLAB includes a large number of features to enable the quick and easy plotting of data. The simplest of these is the x-y plot using the `plot` command. A simple distance, time graph might be produced as follows:

```
>> t = 0:0.5:5
>> distance = [0, 0.2, 0.5, 0.73, 0.74, 1.2, 1.5, 1.6,
1.7, 2.0, 2.33]
>> plot(t, distance, 'o')
```

The third parameter in the plot command is a string specifying the line style, marker type and colour. A table showing these options can be found by typing `help plot` at the command line.

The `plot` command will automatically fit the axes of the figure to fit the data. The `axis` command can be used to control the axis scaling. To set the axes for the current plot use `axis([xmin xmax ymin ymax])`. Further use of the `axis` command can be found using `help axis`.

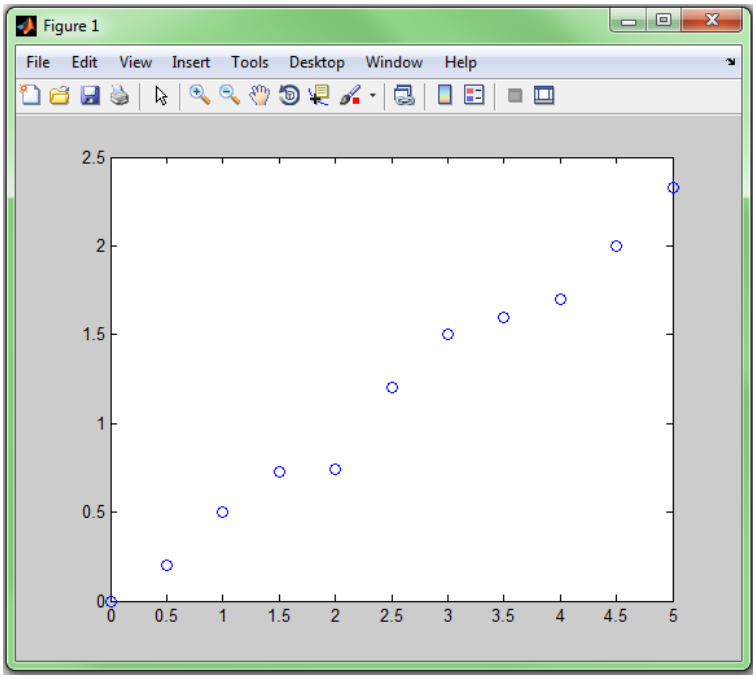

Figure 2. Simple x-y plot

It is also possible to create logarithmic plots using the `semilogx`, `semilogy` and `loglog` commands. Polar plots can be created using the `polar` command and three-dimensional plots created using `plot3`.

An alternative to the plot command is to use data in the Workspace window with the Plots tab: select variable t, ctrl-select variable distance in the Workspace window and then select the plot type required in the Plot ribbon as shown in Figure 3.

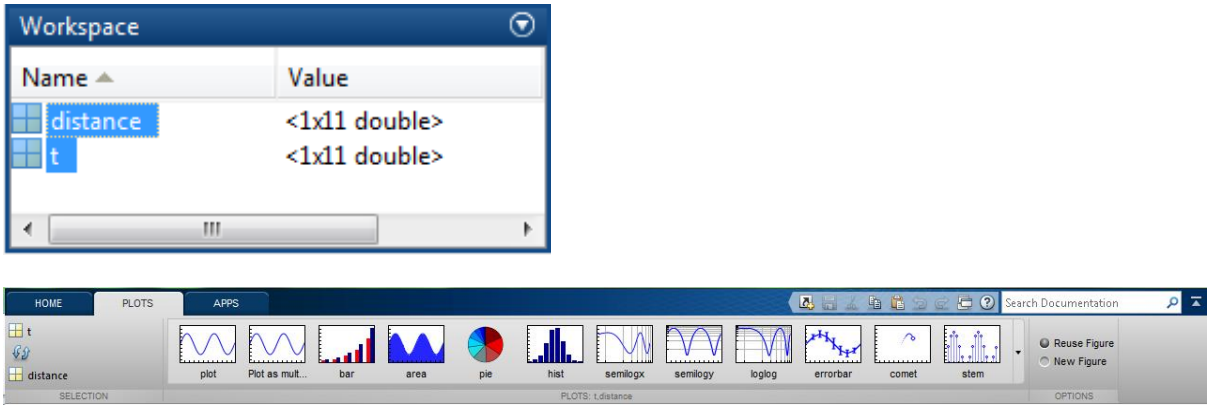

Figure 3. Plotting using the Workspace Window and Plots Tab

Note that the order of selection is important. Selecting distance first would result in a graph with distance along the x-axis. The axes can easily be swapped using the button at the left-hand side of the ribbon.

Exercise 5.1

1) Write a script to calculate the values of the polynomial

   $y = 3x^2 + 4x + 5$

   in the range -10 <= x <= 10 at increments of 0.1.

   Plot the values using a dashed blue line.
2) Run the FreeFall script. Use the Plots tab to create a plot of distance against time.
3) Edit the FreeFall script to create a plot of distance against time, plotting the points using x markers in red.

   Extra task: Set the axes to 10% larger than the data range on all sides.

5.1 Annotating figures

It is usual to label axes and give titles to a graph. This can be done either programmatically or using the plot tools. Adding the code below after a plot command will produce labels as shown in Figure 4.

```
>> xlabel( 'Time, s')
>> ylabel('Distance, m')
>> title('Distance/Time graph')
```

Note that the plot command must be executed before annotations otherwise these will be erased when the plot is created.

To use the plot tools either type `plottools` at the command prompt or select the Show Plot Tools and Dock Figure icon in the Figure window. The Plot Tools window will be displayed as shown in Figure 6. Selecting one of the axes will bring up a dialog in the

Property Editor allowing input of titles and axis labels. Selecting the line displayed will allow the line type, colour etc to be edited in the Property Editor.

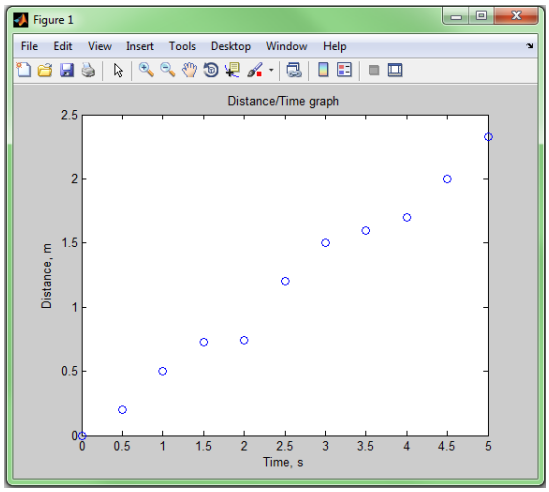

Figure 4. Labelled plot

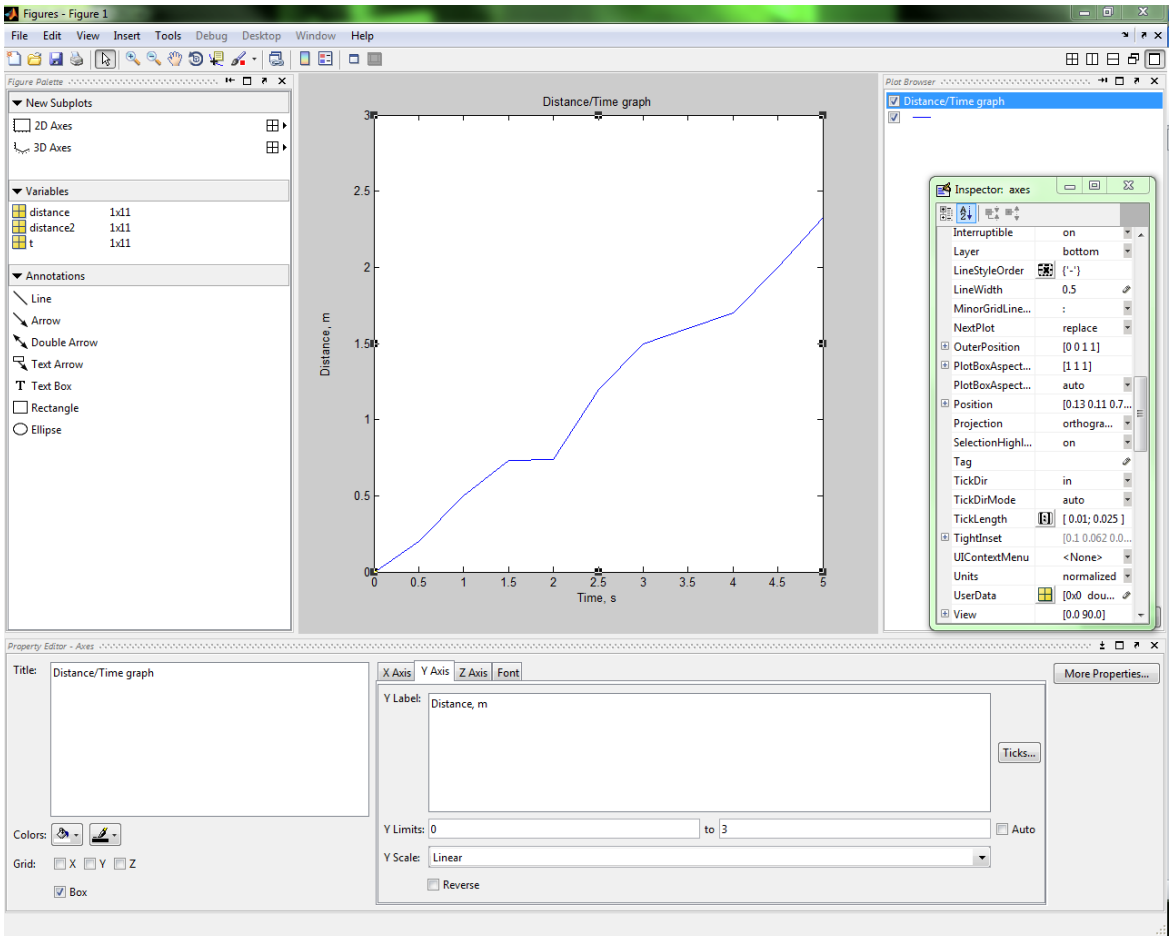

Figure 5. The Plot Tools window

Further changes can be made, eg line attributes, setting axis limits by selecting the More Properties… button. This will open the Inspector window as shown in the Figure 5 where further editing of parameters can take place.

Further information on how to change attributes using the `plot` command can be found by typing `help plot`.

To plot more than one line on a single graph use the `hold on` command. Any subsequent `plot` commands will be plotted on the same figure until the `hold off` command is executed.

Alternatively multiple data sets can be plotted on one figure as follows:

```
>> plot( t, distance, 's', t, distance2, 'o' )
```

or if both sets of distances were collected into one matrix the same result could be achieved as follows:

```
>> distances = [distance;distance2]
>> plot( t, distances, 'o' )
```

In both of these cases the two sets of data will automatically be plotted in different colours. In the second case the same marker will be used for both sets of data.

It is probable that plots will normally be created from within a program. A useful tool is the option of generating code for a plot. This will produce a function which can then be used to create further similar plots programmatically. To do this select `File->Generate Code…` from either the Figure or Plot Tools window.

5.2 Plotting multiple figures

At any time multiple figure windows can be created, each one being identified by a figure number. The current figure number is selected using

```
>> figure(n)
```

All subsequent plotting commands will be performed on the current figure.

5.3 Saving figures

Figures can be saved in various formats using the `saveas` function. `gcf` returns the handle of the current figure which is then used as below

```
>> saveas(gcf, 'filename', 'format')
```

For example

```
>> saveas( gcf, 'Figure', 'png' )
```

will save the figure to a file called Figure.png.

5.4 Subplots

Several graphs can be plotted in the same window using the `subplot(m,n,p)` command. The window is split into a grid of m rows by n columns which are numbered as shown in Figure 6.

p = 1	p = 2
p = 3	p = 4

Figure 6. Ordering for subplot windows

```
% Create a 2x2 grid of plots and plot in bottom right
hand window
>> subplot(2,2,4)
% Call plot commands before calling subplot again for
next location
```

Exercise 5.2

Write a script called Projectiles.m to plot the trajectory of a projectile, launched at an initial velocity of 60 m/s and an angle of pi/3 radians (60 degrees). The equations governing the horizontal and vertical distances travelled are

$h = tV_0\cos(\Theta)$

$v = tV_0\sin(\Theta) - 0.5gt^2$

where t = time in seconds

V_0 = initial velocity in m/s

Θ = launch angle in radians

g = acceleration due to gravity, 9.81 m/s

Label the axes and add a title (to include the launch velocity and angle).

Extra task: Save a copy of the Projectiles script and then adapt it to create a vector of two launch angles and plot both sets of results on one graph with different coloured lines. Change the title to include only the launch velocity and add a legend to show the two angles.

6 Debugging

There are three types of errors which may occur when writing a program: syntax, runtime and logical.

6.1 Syntax errors

Syntax errors are errors in the use of the programming language, for example missing punctuation or misspelt function or variable names.

When entering code at the command line an error message will be displayed:

```
>> disp('Syntax error)

??? disp('Syntax error)

Error: A MATLAB string constant is not terminated
properly.
```

When editing a .m file in the editor window a vertical bar on the right hand side of the window will highlight places where coding errors exist. An orange bar indicates a warning and a red bar an error. Hovering the cursor over the error bar will open a window with a description of the error. In some cases there may be an option to fix the code.

6.2 Runtime errors

Runtime errors occur when the code is executing. An example of this would be trying to access non-existent array members:

```
>> A = [1:3]
A =
     1     2     3

>> for n = 1:4
A(n) = A(n) + 2;
end

??? Attempted to access A(4); index out of bounds because
numel(A)=3.
```

If the error is encountered within a script then the line at which the error occurred will be given as well as the error message.

Dividing by 0 will give a runtime error in most programming languages. In MATLAB it will assign an `Inf` value as the result and continue running the program.

6.3 Logical errors

These are the hardest errors to find. The program may appear to run correctly but may give incorrect results due to incorrect reasoning or other errors such as using an incorrect, but valid, variable.

Two effective ways of finding logical errors in MATLAB code are to use *code sections* or to use the built-in debugger.

6.4 MATLAB Debugger

Using the debugger it is possible to step through code one line at a time, step into functions and to set breakpoints which will stop execution at the point specified. It is also possible to set conditional breakpoints so that the code will only stop when the condition is satisfied.

Breakpoints can be set by clicking in the breakpoint alley just to the right of the line numbers in the edit window. There is a '-' next to lines where it is legal to set a breakpoint. Valid breakpoints will be marked by a red circle. If there is a problem with the breakpoint then the circle will be grey. This could be because the file has not been saved or if there is a syntax error in the line.

Conditional breakpoints can be set by right clicking on a breakpoint and selecting Set/Modify Condition… A dialog will allow a conditional statement to be entered. The code will only stop at the breakpoint when the condition is met.

Having set the required breakpoints it is then possible to run the code stopping at breakpoints and then stepping through line by line as necessary in order to examine values. The debugger commands can be all be accessed in the Debug menu, the Debug toolbar or by using function keys as described below.

Run script (F5) – Commences execution of the file and runs until completion or until a breakpoint is reached.

Step (F10) – Executes the next line of code.

Step in (F11) – Executes the next line of code, stepping into a function if it includes a function call.

Step out (Shift + F11) – Run the remainder of the current function, exit the function and stop at the line after the function call.

Continue (F5) – Continue running either to the next breakpoint or the end of the file.

Exit Debug Mode (Shift + F5) – Stops running the file and exits.

Run to Cursor – Access this either using the button or by positioning the cursor at the required line, right click and then select Go Until Cursor. The file will then be executed until this point is reached.

At any point the current values of any variables will be available to be examined in the workspace. Also the values of variables can be shown in a similar way to a tooltip by holding the mouse over the variable in the editor window.

Exercise 6.1
Either use the debugger to debug your own Projectiles script or try debugging ProjectilesBuggy.m

6.5 Code Sections

The use of code sections enables a .m script file to be split into logical sections which may then be executed one by one. This allows the results of that section to be examined before continuing to the next section.

A code section boundary is defined by inserting a line that begins with a section break, %%. Text may follow the section break providing there is white space after the %% characters. Typically this will be a title for the section (this will be used by the Publish feature described in Section 14). Selecting the Insert Section Break button in the toolbar will automatically insert a section break at the current cursor position.

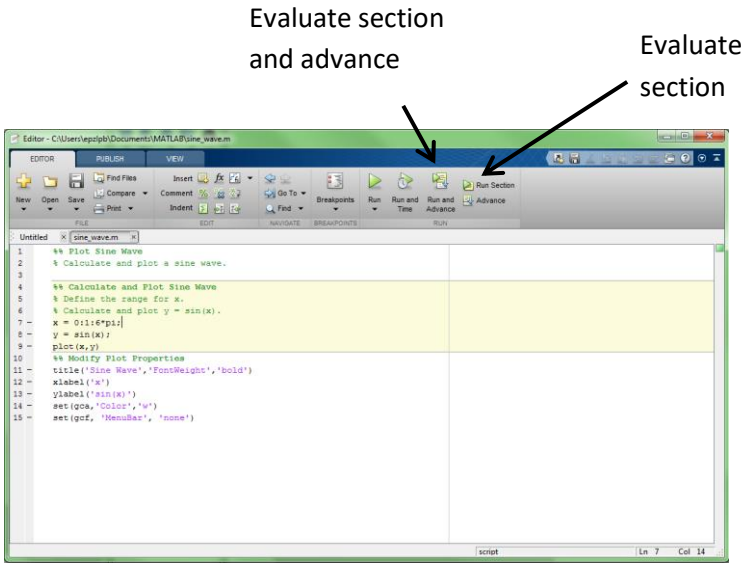

Figure 7. Code sections

The current section is governed by the cursor position and is highlighted as shown in Figure 7. The Run Section button will run the current section. The Run and Advance button will automatically change the current section to the next section.

Exercise 6.2
Add code sections to the Projectiles script.

7 Program Structure

So far we have only considered simple programs where the flow through the set of instructions is sequential. More commonly the path through the program will be more complex; either selection may be made based on logical conditions or a section of the program may be repeated in a loop. This is illustrated in Figure 8.

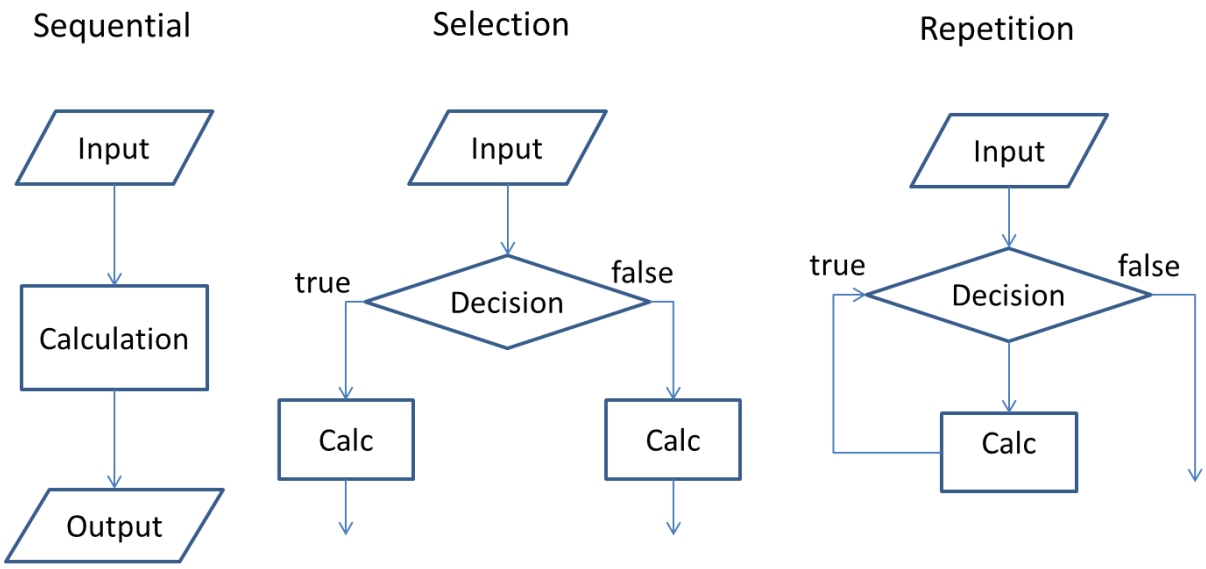

Figure 8. Program constructs

It is important to plan the program, typically either using a flowchart or by writing pseudocode. When writing pseudocode the outline of the program is described using a series of statements. If these are typed directly into an M-file then they can often be turned into comments and the actual MATLAB code can be inserted between them. The following sections describe the constructs within MATLAB which allow more complex code to be created.

8 Conditional Operators

Where a branch is to be made in a program, the decision as to which route to take will be based on the result of either a relational or logical operation.

8.1 Relational and logical operators

The logical data type in MATLAB can be assigned to be *true* or *false.* The following relational and logical operators produce a result with a logical data type. In MATLAB these are as follows:

Relational	`a < b`	less than
	`a <= b`	less than or equal to
	`a > b`	greater than

	`a >= b`	greater than or equal to
	`a == b`	equal to
	`a ~= b`	not equal to
Logical	`a & b`	and
	`a | b`	or
	`~ a`	not
	`xor( a,b )`	exclusive or

The *xor* operator will return true if one, and only one, of the expressions is true. For example

`>> xor( 3 > 1, 7 > 10)` will return a value of 1 whilst

`>> xor( 3 > 1, 7 < 10)` will return a value of 0 as both arguments are now true.

Care should be taken when comparing values of floats or doubles. Small differences in values may result in incorrect results if checking for equality. Instead use a statement similar to this:

```
>> abs(a-b) < tolerance
```

which will check that the absolute value of the difference between the values is smaller than a given tolerance

8.1.1 Logical Indexing

If these operators are used on matrices the operation is performed element-wise and the result will be a matrix with one result for each element, as shown by B in the following example.

```
>> A = [1:5;3:7]
A =
     1  2  3  4  5
     3  4  5  6  7

>> % Create logical matrix with result of comparison
>> B = A > 2 & A < 6
B =
     0  0  1  1  1
     1  1  1  0  0
```

This matrix can then be used as a mask to perform operations on selected elements of a matrix.

```
>> Use mask on left hand expression to only change values
>> set to true
>> A(B) = sqrt(A(B))
A =
     1.0000    2.0000    1.7321    2.0000    2.2361
     1.7321    2.0000    2.2361    6.0000    7.0000
```

To create a separate vector containing only the elements which satisfy the test the mask can be used as follows:

```
>> C = A(B)
C =
     3
     4
     3
     5
     4
     5
```

There are several built-in functions in MATLAB which can be used with logical matrices. The `find` function will return the indices of the matrix which satisfy the test. For the original matrix A above this could be used as below.

```
>> D = find(A>2 & A<6)
D =
     2
     4
     5
     6
     7
     9
```

Taking `A(D)` will have the same result as in matrix `C` above. In either case a further operation, such as finding the square root, could then be performed on the elements which fulfil the test criteria. These steps could be combined into a single statement giving

```
>> E = sqrt( A(find(A>2 & A<6)) )
E =
    1.7321
    2.0000
    1.7321
    2.2361
    2.0000
    2.2361
```

Note that if the above command is written in the editor window a red wavy line will appear under the `find` command. These give good tips for improving the performance of code written in MATLAB. In this case it shows that performance can be improved by using logical indexing directly.

```
>> E = sqrt(A(A>2 & A<6))

E =
    1.7321
    2.0000
    1.7321
    2.2361
    2.0000
    2.2361
```

Using the form `I = find(X,1)` will return the linear index of the first element in the array which is nonzero, or which satisfies the condition if a logical operator is used.

```
>> B = [0 0 0 3 4 5 6]
B =
     0     0     0     3     4     5     6
>> find( B,1 )
ans =
   4
>> find( B>4, 1)
ans =
   6
```

The `find` command can also be used to return the row and column indices of a matrix

```
>> A = [1:5;3:7]
A =
     1 2 3 4 5
     3 4 5 6 7

>> [row, column] = find(A>3 & A<6)
row =
     2
     2
     1
     1


column =
     2
     3
     4
     5
```

Other functions are `any`, which returns `true` if any element of the matrix is non-zero and `false` otherwise, and `all` which returns `true` if all elements are non-zero. The `isequal` function can be used to compare all elements of two matrices and returns `true` if all elements are equal, otherwise `false`.

Exercise 8.1
Create the following matrix in the workspace

```
A = 9  12  18  0
    10  3   1   7
    2   5  14  22
```

1) Create a logical matrix B giving the elements of A which are greater than or equal to seven and less than 15.
2) Modify matrix A so that each element of A which satisfies the condition in (1) is squared.
3) Find the linear indices of the revised matrix A which are greater than 100.
4) Find the values of A which are greater than 100.

8.2 Conditional Statements

The results of relational and logical operations can be used as the comparison for conditional statements.

8.2.1 If Condition

A simple `if` condition can be used if a set of command are to be executed only if a certain condition is satisfies. The syntax for the `if` statement takes the form

```
if condition
     Code block
End
```

`if` statements can be nested as follows:

```
     if condition1
          Code block 1
          if condition 2
               Code block 2
          end       % End of condition 2
          Code block 3
     end            % End of condition 1
```

It is not necessary to indent the code blocks within the `if…end` statement but it is good practise and leads to much more readable code, particularly where nested as above.

8.2.2 If/Else Condition

The `if/else` construct is used where one set of instructions is to be executed if a condition is true and an alternative set executed if the condition is false. The syntax takes the form

```
if condition 1
      Code block 1
else
      Code block 3
end
```

Note that there is one `end` statement for the whole construct.

Exercise 8.2

Change the Projectiles script to take the angle as user input and add range checking to ensure that the angle entered is within the range 0 to 90 degrees. Output an error message if the value is out of range, otherwise continue and calculate values.

Extra task: Modify the script to only plot values with positive vertical displacements.

8.2.3 If/Elseif/Else Condidtion

Where there are several sets of code to be executed depending on multiple conditions then the `if/elseif/else` construct can be used. There may be several `elseif` statements but only one `else` and one `end` to complete the construct. The syntax takes the form

```
if condition 1
      Code block 1
elseif condition 2
      Code block 2
else
      Code block 3
end
```

Exercise 8.3

Write a program to assign a class of degree pass with input based on the table below. Output a message to say which class is awarded. Valid inputs are marks between 0 and 100.

Class	Mark
First	>= 70
Upper second	60 to 69
Lower second	50 to 59
Third	40 to 49
Fail	< 40

Choose a set of values as test data which will test all paths through the program.

8.2.4 Switch Statements

The switch statement is a useful construct which can be used as an alternative to a set of `elseif` statements. The function of the switch statement may be replicated by an if/elseif/else statement but the switch statement may be clearer to read. Note that the different blocks are executed depending on the value of one given variable which may be either a number or a string.

```
switch variable
      case value1
            Code block 1
      case value2
            Code block 2
      …
      case value n
            Code block n
      otherwise
            Code executed if expression none of
            values specified
end
```

A useful example of where a switch statement might be put to good use is the menu option. The following piece of code displays a menu and returns a scalar value depending on the button pressed. A switch statement is then used to process the selections.

```
%Menu and switch/case statement example
Colour = menu('Select Colour', 'Red', 'Blue', 'Green');
switch Colour
    case 1
        disp('You chose red');
    case 2
        disp('You chose blue');
    case 3
        disp('You chose green');
    otherwise
        disp('You didn''t make a selection');
end
```

9 Repetition Operators

Where a line or section of code needs to be repeated several times then loop operators can be used.

9.1 For Loop

The `for` loop is used to iterate through a set of values specified by a vector or matrix which are used to execute the commands within the loop. It is used when the number of times that the code is to be executed is known. The syntax is as follows

```
for index=[matrix]
     Code block
End
```

It should be stressed that `for` loops will be required much less frequently in MATLAB programs than in other languages. If you intend to use a for loop in your code stop and think about whether it is necessary or whether it could be replaced with an array operation or function.

9.1.1 Nesting loops

Note that, as `for` loops are largely unnecessary in MATLAB, nesting of loops is also almost always unnecessary. However, the format for nested `for` loops is shown below. Note the completion of each loop with the `end` statement.

```
for index1 = [matrix1]
  Code block 1
  for index2=[matrix2]
      Code block 2
  end
end
```

Exercise 9.1

Write a script to convert degrees Fahrenheit to degrees Celsius using the equation

$$T_{°\mathrm{C}} = (T_{°\mathrm{F}} - 32) \times \frac{5}{9}$$

Use code sections to write three versions of the calculation using integer values from 1 to 1000 as input. Use tic/toc to time each version of the loop.

1. Using a `for` loop without pre-allocating space for the results
2. Using a `for` loop but pre-allocating space first
3. Using vectorisation

9.2 While Loop

Where a set of commands are to be repeated but the number of times they are to be executed is unknown a `while` loop is used. The loop is executed until a given condition is met. A typical use

of this would be an iterative method which would continue until a value is within a given tolerance. The format for the `while` loop is

```
while condition==true
      Code block
end
```

Sometimes it is necessary to terminate a loop early. An example might be if the number of iterations in an iterative method exceeds a maximum. The `break` statement can be used to exit the loop. Sometimes the execution of the current loop needs to be skipped. In this case the `continue` statement can be used. This skips to the start of the next loop. Both of these can be used in either `for` or `while` loops.

Exercise 9.2

Change the Projectiles script to loop until valid inputs have been entered. Select an appropriate set of values to test the program.

10 Functions

Sections of code which are used repeatedly can be separated from the main body of the script and used to create a function. If the folder in which functions are stored is added to the path as described in Section 1 then the function will be accessible from any other script.

Functions in MATLAB are created in a separate .m file. The file name and function name must be identical.

The general format of a function is

```
function [argout1, argout2..] = funcName( argin1, argin2,..)
% H1 comment line
% Other comments
Code Block
(return)
(end)
```

The input and output arguments in the function declaration are dummy arguments which act as placeholders for the actual arguments which are passed to and from the function. The return and end statements are optional.

The following very simple function calculates the distance between two points

```
function distance = DistBetweenPoints( x1, y1, x2, y2 )
% DistBetweenPoints - Function to calculate the distance
between
% two cartesian points (x1,y1) and (x2,y2)
%
% Function returns the distance between the points
% Format:
```

```
% distance = DistBetweenPoints( x1, y1, x2, y2 )

%Calculate distance

distance = sqrt( (x2 - x1)^2 + (y2 - y1)^2 );

end
```

Care must be taken when writing MATLAB functions to ensure that the function will work if arrays are passed as parameters. If arrays, rather than single values, are passed to the DistBetweenPoints function as above the following error will be generated.

```
??? Error using ==> mpower
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.

Error in ==> DistBetweenPoints at 10
distance = sqrt( (x2 - x1)^2 + (y2 - y1)^2 );
```

To ensure that functions will operate on arrays of values it is essential to use the . operators within the function. Simply changing the line which calculates the distance to

```
distance = sqrt( (x2 - x1).^2 + (y2 - y1).^2 );
```

will resolve the issue and calculate distances for sets of pairs of points.
If more than one parameter is returned from the function then the format for calling the function would be

```
[output1, output2] = Function(input1, input2);
```

Exercise 10.1
Create a function to convert degrees Celsius to Fahrenheit using the equation

$T_{(°F)} = T_{(°C)} \times 9/5 + 32$

Input: temperature in degrees Celsius
Return value: temperature in degrees Fahrenheit
Write a script to test the function for input values of 0 and 100 and also for a vector of values.

Extra task: Write a function to calculate the values of the polynomial $y = 3x^2 + 4x + 5$ (you could adapt the script written in Exercise 5.1)
Input: x
Return value: y
Write a script to test the function for input values of 0 and 10 and also a vector of values.

10.1 Subfunctions

It is permissible to store more than one function in a function M-file. The first function with the same name as the file is the primary function and subsequent functions are called subfunctions. Subfunctions are only visible to other functions in the same file. They are declared in the same way as the primary function and may be in any order as long as the primary function is first in the file.

The help for subfunctions is provided by the H1 line in the same way as the primary function. It is accessed by the filename of the primary function without the .m extension and the > symbol. For example a subfunction, mySubFunc, in file myFunction.m would be displayed by typing

```
>> help myFunction>mySubFunc
```

10.2 Anonymous functions

MATLAB allows the creation of local functions called *anonymous functions.* These are created in the command window or in a script file and are available only until the workspace is cleared. They do not need to be stored in a .m file. The syntax for anonymous functions is

```
Fnhandle = @ (input arguments) functioncode;
```

The `@` symbol shows MATLAB that a function handle is being created which provides a way of referring to the function. `(x)` gives the parameters passed to the function and then the function is defined. An example might be

```
ln = @(x) log(x)
```

As with normal functions care should be taken to use the . operators so that arrays may be safely passed to the function.

The anonymous function may be saved in the same way as other data using the load and save commands.

10.3 Function functions

Function functions are functions which take another function as input. An example of this is the fplot command which allows a function handle to be passed as one of the parameters. For example the fplot function can be sent the handle of the `sin` function as a parameter using the @ symbol. For example

```
fplot( @sin, [-pi,pi] ); %plot sin in range –pi to pi
```

The advantage of the use of function handles can be seen in the case where a polynomial expression is to be sent to a function. For example an anonymous function

`polyn = @(x)2.*x.^3+4.*x.^2+5.*x+7` can be created and then the function handle, poly, used as an input to the plot function. This is much neater than sending the whole polynomial as a parameter.

`plot( polyn( 0:10 ) )` will plot values of the polynomial from 0 to 10.

10.4 Persistent variables

Normally all local variables are cleared when a function terminates. In some cases it may be useful to retain a value until the next time a function is called, for example to count the number of times that an error has occurred. In this case a persistent variable may be declared as follows

```
persistent count
% Check if count has been initialised and set to 0 if not
if isempty( count )
     count = 0;
end
count = count + 1;
```

The first time a function containing this code is called the variable count is set to 0 and then incremented to 1. On subsequent calls the value of count will be retained from the previous call and then incremented by 1.

Persistent variables may be cleared from memory, and hence restarted, by using the `clear functions` command.

Exercise 10.2

Create an anonymous function `poly1` for the expression

$y = 3x^2 + 4x + 5$

Use the anonymous function and the `plot` function to plot the function in the range -10 to 10.

Use the `fplot` function to plot the function in the range 0 to 20.

Calculate the values of the function from 0 to 10.

11 Tables

Tables can be used to store heterogenous, column oriented or tabular data. Variables can have different data types but all columns must have the same number of rows of data. Data is not restricted to column vectors but matrix data must also meet the requirement for consistent numbers of rows.

Tables can be created from either workspace data or loaded from data in text files.

Using the `table` function:

```
TableName = table(var1, var2, var3,… );
```

or the `readtable` function to load a data set:

```
TableName = readtable('data.dat');
```

Whole columns of data can be accessed using dot notation and the column name. For example,

```
PatientData.Gender  % Accesses the whole column
PatientData.Gender(10) % Accesses data in column
```

Particular rows and columns can be accessed using subscript notation:

```
PatientData(1:3, :);
ans =
    Gender      Age    Height    Weight
    ________    ___    ______    ______
    'Male'      38     71        176
    'Male'      43     69        163
    'Female'    38     64        131
```

Extra data can be added by using a new column name and assigning it a data item with the correct number of rows, for example:

```
PatientData.ID = (1:100)';
```

Columns can be deleted using []:

```
PatientData.ID = []
```

Table data can be written to a file using the `writetable` function:

```
writetable( PatientData, 'PatientData.txt');
```

11.2 Table Metadata

Metadata can be associated with a table, stored in the table properties. The metadata fields available are variable descriptions, variable units, dimension names, user data and row names. The metadata can be accessed and edited using the Properties label and field name:

```
% Access the metadata

PatientData.Properties.VariableNames

ans =

     'Gender'    'Age'    'Weight'    'Height'

% Set the metadata variable units field

PatientData.Properties.VariableUnits{'Height'} = 'inches'
```

11.2 Indexing Tables using Row and Variable Names

If the RowNames property has been set this can then be used as an index into the table. The row names strings must be unique.

```
% Set the row names property fields

PatientData.Properties.RowNames = LastName;
```

Data can then be selected by row name. Note that the row names are assembled into a cell array for use as the row parameter if more than one row is to be accessed. For example

```
PatientData({'Wilson', 'White'}, : )
```

Selections can also be made using the variable names:

```
PatientData('Russell', {'Height','Weight'} )

ans =

           Height    Weight

           ______    ______

Russell    69        188
```

11.2.1 Creating Cell Arrays for String Variables

Cell arrays are a data type which can store different types of data which can then be accessed by index. One advantage of cell arrays is the ability to store data of different lengths, particularly strings. A normal array of strings will require the strings to be padded with spaces to ensure that they are all the same length. This is not necessary with a cell array.

Cell arrays are created using {} brackets. To create a cell array consisting of a number of string variables send the strings as input to the array:

```
CellArray = {'Age', 'Weight', 'Height'}

CellArray =

     'Age'    'Weight'    'Height'
```

Exercise 11.1

Load the patients data and then create a table using the LastName, Smoker, Weight and Height data. Plot a graph of weight vs height using a blue cross to mark data for smokers and a red circle for non-smokers.

Add axis labels and a legend.

12 Data Import and Export

Data can be imported from various file formats including text files and spreadsheets. Typing `doc fileformats` will give a complete list of supported file formats. Importing of data can be carried out either interactively using the *Import Wizard* or programmatically.

12.1 Importing Data Using the Import Wizard

The Import Wizard can be invoked by right clicking on the filename to be imported and selecting Import Data… or using the ImportData button. The dialog will display the numerical data as read in from the file, along with any row or column headers as shown in Figure 9.

How the data is to be imported can be selected from the Imported Data menu in the centre of the toolbar.

In the case shown here two 30x1 arrays are selected, one for each of the x and y column names. If the 'Matrix' option had been selected then a single 30x2 array would have been formed. Double clicking on the variable names at this point allows them to be renamed.

Selecting Finish will complete the operation and the data will appear in the Workspace.

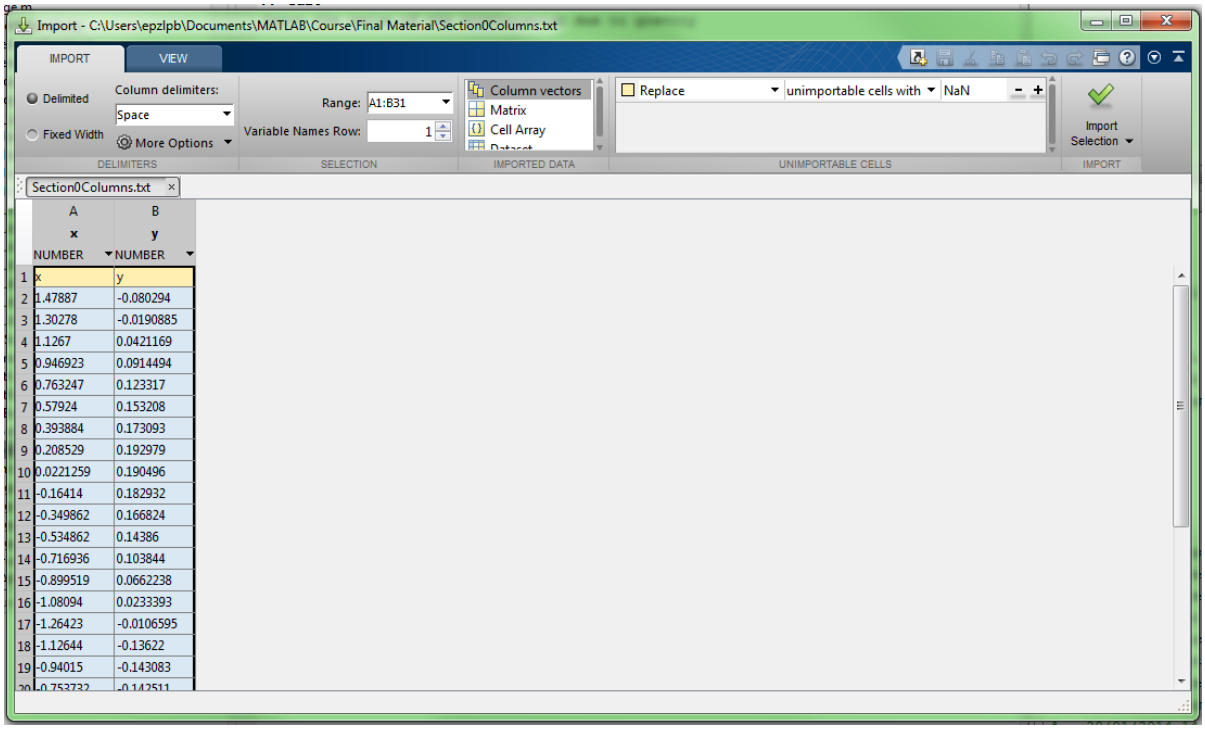

Figure 9. Import Wizard

The Import Wizard can also be launched from the command line using the `uiimport` function as follows:

```
>> uiimport('filename.extension')
```

For example an image file could be imported with the command

```
>> uiimport('SimonVega.jpg')
```

resulting in the Import Wizard dialog shown in Figure 10.

Note that the image has been imported as a three dimensional array: for each of the 1936x2592 pixels in the image there is an array of three 8-bit unsigned integers (as indicated by the class uint8), representing the RGB data in the image.

Ticking the Generate MATLAB code button on the dialog will generate a .m file with a function which will replicate the file import. This can then be saved and used to import other, similar, files from within a script.

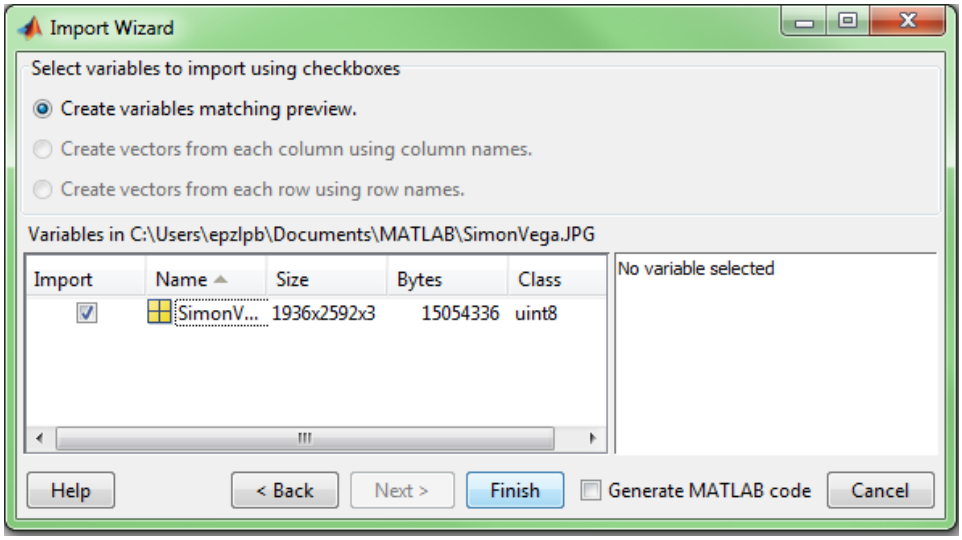

Figure 10. Importing an image using the Import Wizard

The data can now be manipulated in the same way as any other MATLAB array. For example the following commands will change the image as shown in Figure 11, giving a 100x100 square with either the red byte set or all three set to give a white square.

```
% Change the red component to 255
>> SimonVega(100:200, 100:200, 1) = 255;
% Change R,G and B components to 255
>> SimonVega(200:300, 200:300, :) = 255;
```

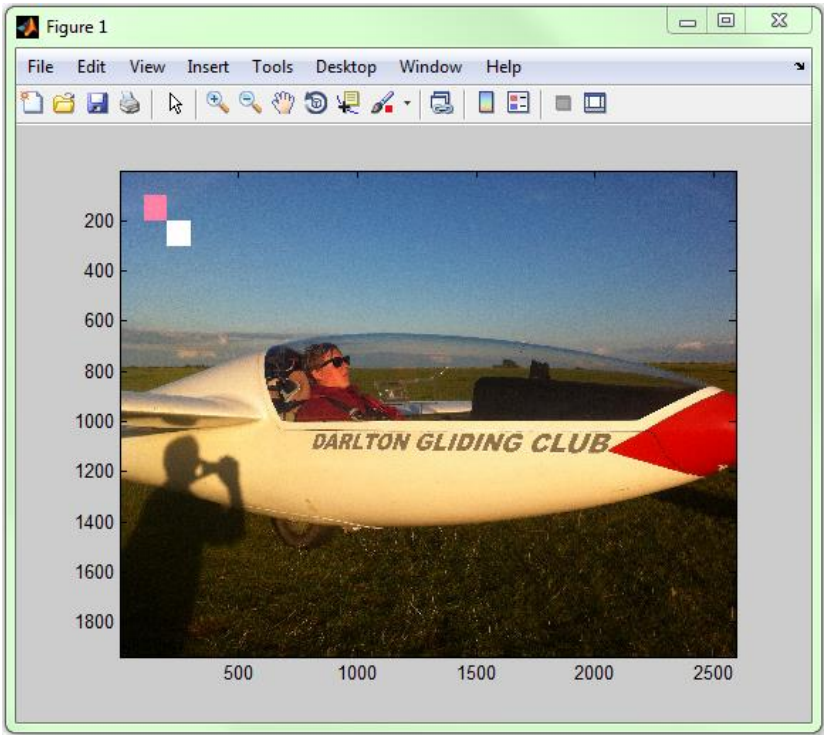

Figure 11. Imported image data after editing

12.2 Cleaning Data

Missing or bad data items will be converted to NaN, 'Not a Number' by MATLAB on import. Logical indexing can be used to extract these items so that only good data is processed. The `isnan` function will return a true value for any item in the data sent to it with a NaN value. A small case is shown below

```
>> data = [0 2.3 5.6 NaN 5 7 NaN -2]

data =

         0   2.3000   5.6000    NaN    5.0000     7.0000     NaN    -
2.0000

>> badData = isnan(data)

badData =

     0     0     0     1     0     0     1     0

>> CleanData = data(~badData)

CleanData =

         0    2.3000    5.6000    5.0000    7.0000   -2.0000
```

Exercise 12.1

Create a function using the Import Wizard to import age and BMI from MedicalData.txt. Create a script which will call the function to import the age and BMI data, clean the data and then create a plot of age vs BMI for the first 1000 items of data. Plot the points with a 'o' marker, labelling the axes and adding a title.

13 Publishing

MATLAB includes the facility to publish code into various formats, eg Word, html, Latex. The publish feature uses code sections to separate the document into sections and uses the text after the %% cell boundaries as titles for the sections. The initial H1 comment is used as a title for the document.

A file can be published either using the Publish button on Publish tab of the editor window. This will use the default configuration. The configuration can be edited by selecting the down arrow on the Publish button and selecting Edit Publishing Options… . This allows the file type and various other settings to be configured. It also allows input arguments to be specified for files requiring input. Note that programs requiring user input cannot be published.