

# 1 Getting Started with MATLAB

This tutorial introduces the MATLAB environment. It is intended as a step-by-step guide for those new to working with MATLAB, and a recap for those with some prior experience.

The tutorial comprises a set of exercises, each of which ask you to follow a series of steps. Once you have worked through all of the exercises, you should have an understanding of the following:

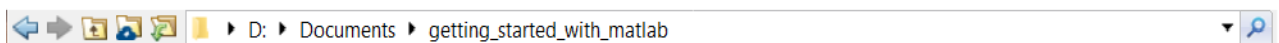
- How to use the MATLAB Command Window and Workspace
- How to access the available help and documentation
- How to create, save, and run MATLAB scripts and MATLAB functions
- How to generate basic plots
- How to work with different data types in MATLAB
- How to work with sampling rates

MATLAB has many more features than can be covered here, so we focus on the core aspects.



## 1.1 Introducing MATLAB

The aim of this first exercise is simply to demonstrate the MATLAB interface, and highlight some important features.

It is assumed that you have already opened MATLAB. MATLAB usually opens in either the last working directory or the install location. The location of the working directory can be seen in a bar towards the top of the MATLAB environment, as shown in Figure 1.1.



**Figure 1.1:** The current MATLAB working directory

You can change the current working directory by  on the  icon, on the right hand side of the working directory bar. Choose a suitable location now. Note that the directory path must **not** contain spaces.

---

**Exercise 1.1 MATLAB Orientation and Using the Command Line**

In this first exercise, we will take a quick tour of the MATLAB interface, and point out important features. Note that the screen-shots provided show MATLAB 2020a — if using a prior or subsequent version, the appearance may differ slightly.

- (a) **Explore the MATLAB interface.** When MATLAB is open, you should see an environment and layout similar to that shown in Figure 1.2.

The MATLAB interface comprises a number of panes which are highlighted in the diagram.

- (b) We can execute commands by typing them directly into the Command Window after the MATLAB prompt (`>>`) and pressing the **Enter** key.

Perform a simple addition of two numbers by entering the command:

```
>> 3 + 9
```

Apart from outputting the correct answer, you should notice a new variable has been created in the Workspace panel called `ans`. This variable is created automatically whenever an output variable has not been specified and will update with the result of the latest command not to be assigned an output variable.

- (c) We can assign the calculation a variable by using the `=` operator:

```
>> a = 3 + 9
```

You will see that the variable `a` has appeared in the Workspace panel. MATLAB variables can be named anything as long as they **start** with a letter and contain only letters, numbers, and underscores (`_`). MATLAB variables are also case sensitive.

- (d) **Create a second variable.** Create the variable `b`, and assign it with the value 4. Notice the semi-colon at the end of the statement:

```
>> b = 4;
```

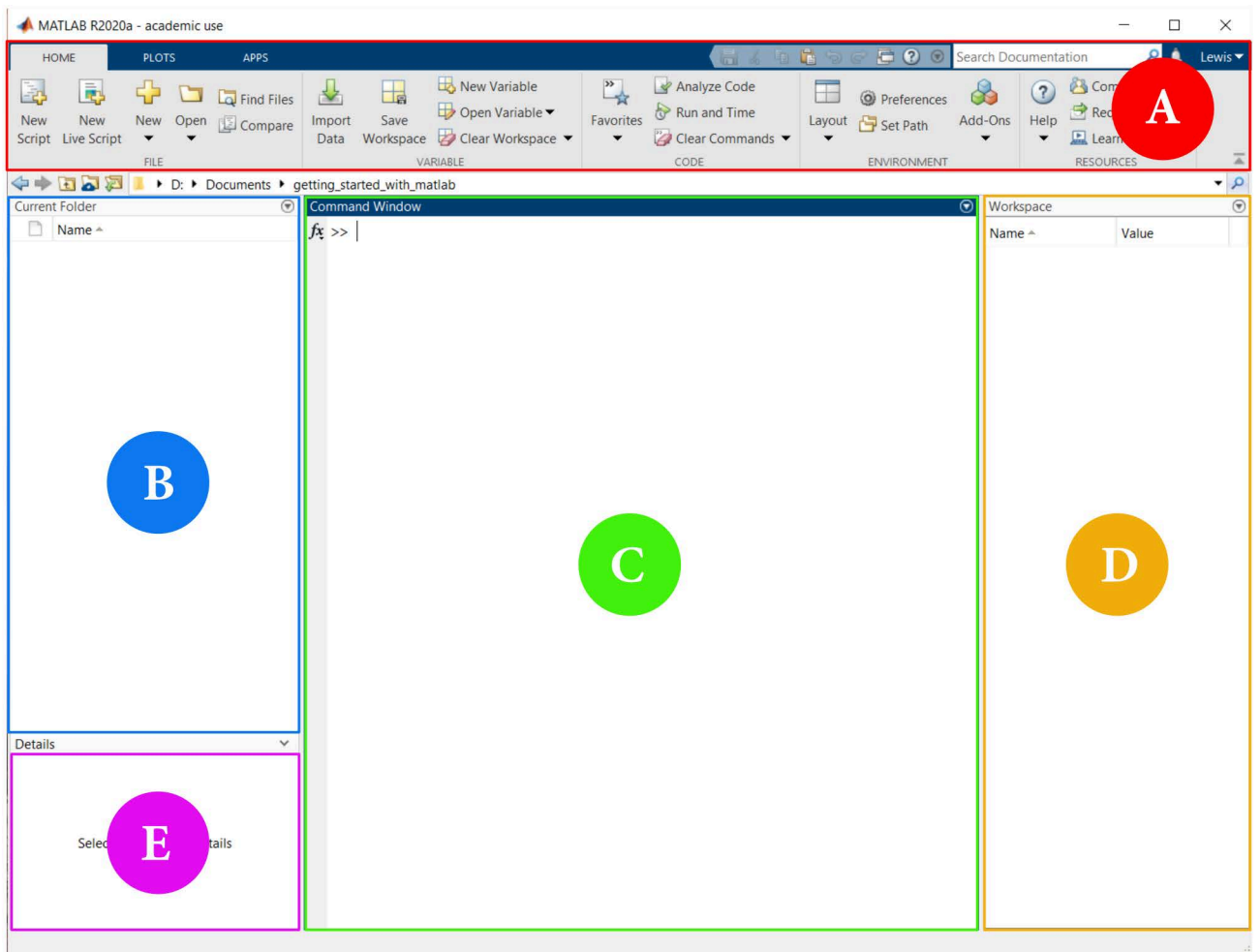
Entering a command with a semi-colon suppresses the result from being displayed. Although the result will not appear in the command window, the value can still be seen in the Workspace.

- (e) **Perform arithmetic operations.** Next, try finding the product of `a` and `b`, by typing:

```
>> a * b
```

Detailed below are some common operators, give them a try:

Addition (+), Subtraction (−), Multiplication (\*), Division (/), Power (^).



- A** The **Menu Bar**, shown here with the main 'Home' tab open. This allows you to work with files and variables, open Simulink, setup the environment, and access help and support.
- B** The **Current Folder** pane shows the files within the present folder. This facility acts like Windows Explorer — you can open files, as well as moving, renaming and deleting them.
- C** The **Command Window** is the place for directly entering commands, including single statements or short sets of statements, running scripts and calling functions.
- D** The **Workspace** shows all of the variables currently held in memory, of various types (scalars, arrays, matrices, strings, structures, etc.). These can be inspected / edited here.
- E** The **File Details** pane simply displays a summary of the file currently selected in the *Current Folder* pane. This includes revision history details and a description / preview.

**Figure 1.2:** The MATLAB environment, shown with default layout

- (f) **Use a function.** Next, enter the following code to find the remainder when *a* is divided by *b*. Notice that this code calls the function `rem()`, which is an inbuilt function of MATLAB.

```
>> r = rem(a,b)
```

- (g) Try assigning `a` and `b` with different values, and recalculating the value of `r`.
- (h) **Obtain help on a function.** MATLAB and its toolboxes provides a large number of functions that users can leverage in their code. It is easy to obtain information about a particular function. To find out more about the `rem()` function, type:

```
>> help rem
```

You should now see details of the `rem()` function appear within the command window:

```
>> help rem
rem      Remainder after division.
rem(x,y) returns x - fix(x./y).*y if y ~= 0, carefully computed
to avoid rounding error. If y is not an integer and the quotient
x./y is within roundoff error of an integer, then n is that
integer. The inputs x and y must be real and have compatible
sizes. In the simplest cases, they can be the same size or one
can be a scalar. Two inputs have compatible sizes if, for every
dimension, the dimension sizes of the inputs are either the same
or one of them is 1.

By convention:
    rem(x,0) is NaN.
    rem(x,x), for x~=0, is 0.
    rem(x,y), for x~=y and y~=0, has the same sign as x.

Note: MOD(x,y), for x~=y and y~=0, has the same sign as y.
rem(x,y) and MOD(x,y) are equal if x and y have the same sign,
but differ by y if x and y have different signs.

See also mod.

Documentation for rem
Other functions named rem
```

Noting the last point in this Help text (the last two lines in the previous box), i.e. that you enter:

```
>> doc rem
```

...at the command prompt, to view the same information in a browser. Try this now and confirm that a window opens showing the desired help information. You should also notice that some examples are provided to demonstrate how to use this function. As shown in Figure 1.3, the contents menu on the left hand side, and the search facility at the top of the window, can be used to find information about other functions if desired.

Try finding out about some other functions, such as `mod()`, `power()`, and `ceil()`.

Accessing information about functions using the internal documentation is beneficial as it relates directly to the version of the tool that is being used. Whereas online documentation is provided for the latest version of the tool.

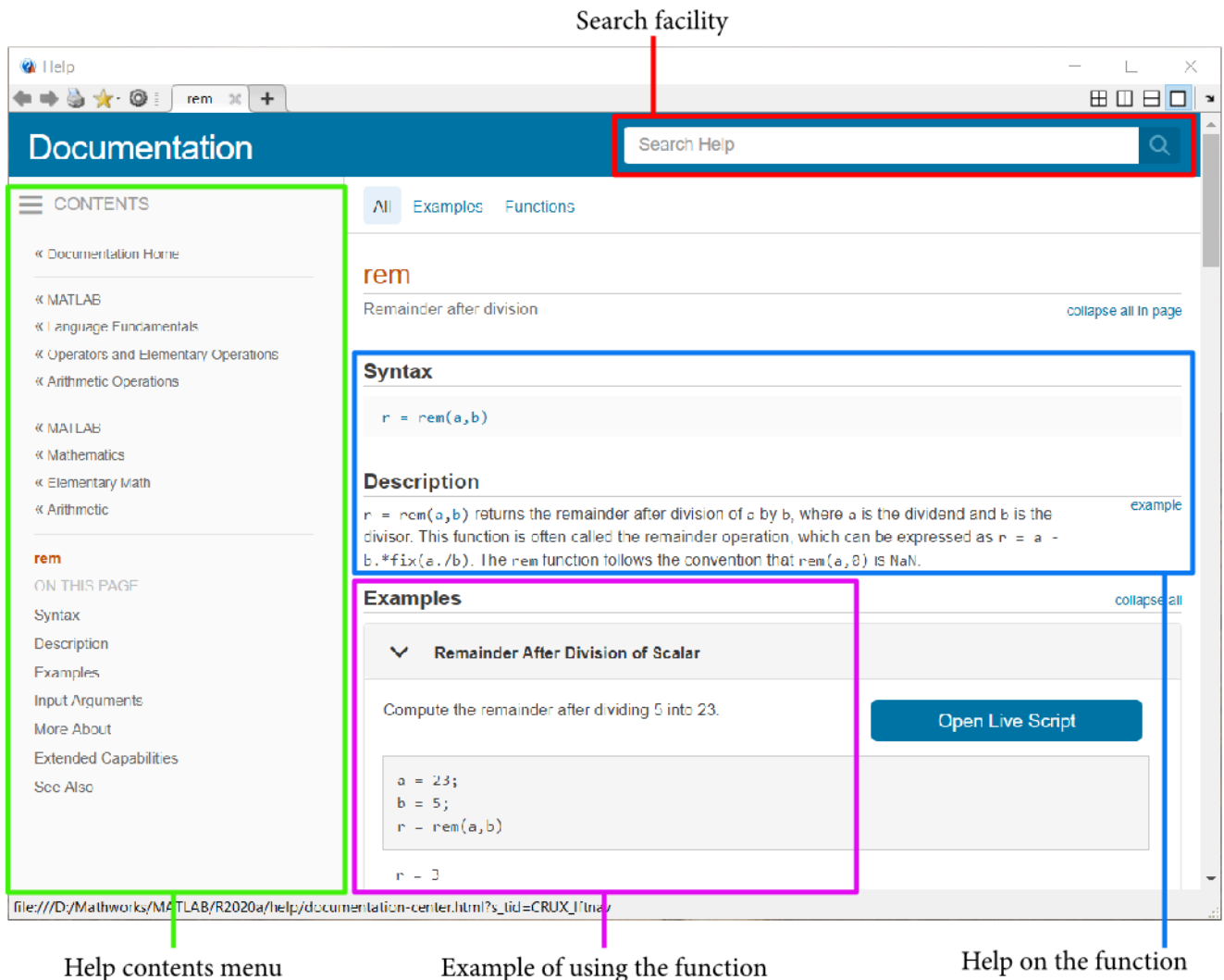



Figure 1.3: Layout of the help browser (typical view)

- (i) **Using the command history.** MATLAB keeps a memory of commands executed at the command prompt. This can be used to view recently executed commands, and it also provides a quick and easy way to repeat a previous command (either directly, or with modifications).

Place a cursor at the command prompt and press the **UP** arrow key on your keyboard. This will cause a list of previous commands to be displayed (the command history). You can now use the **UP** and **DOWN** arrow keys to highlight previous commands from a list. As you change line, you should notice that the highlighted code appears at the command prompt as shown in Figure 1.4.

If you press the *Enter* key, MATLAB will execute the code currently shown at the command line. Alternatively, you can  on the command line, and edit the code before pressing *Enter* to execute.

- (j) **Another command history tip!** If you would like the command history to form a subset of commands that begin in a certain way, type the first character (or few characters) before pressing the UP key. For instance, if you type 'r' before pressing UP, the command history will allow you to select only those commands that begin with r, as shown in Figure 1.5. This can make it quicker to retrieve a previously used command.

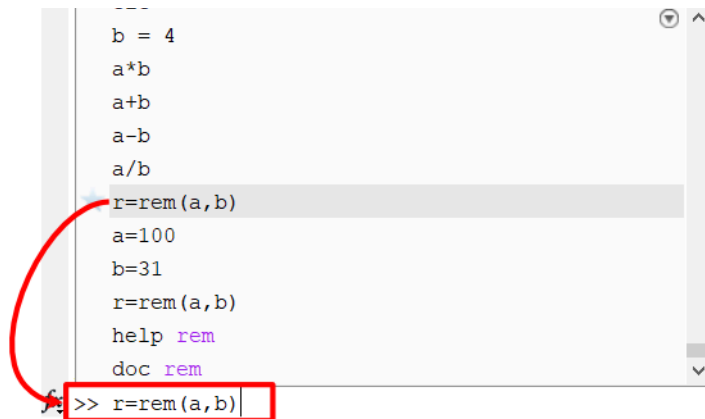


Figure 1.4: Command history

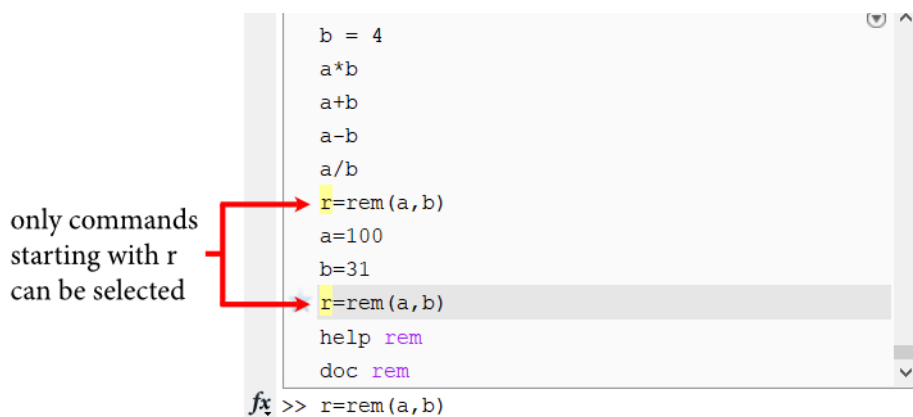


Figure 1.5: Selective command history

- (k) **Inspecting and editing Workspace variables.** As noted earlier, variables and their values appear in the Workspace panel of the MATLAB interface. In addition to being useful for inspecting variables and values, the Workspace pane can be used to change the value of a variable directly (i.e. without writing code).

To do this, **2** on the variable (name or value) in the Workspace. This will open a new pane showing the details of the variable, similar to that shown in Figure 1.6 (in this case it is just a single value, but in other circumstances there may be an array or matrix of values). Next, **2** in the desired cell until a cursor appears, edit the value, and press *Enter* to finish. You should see the value change in the Workspace pane as well.

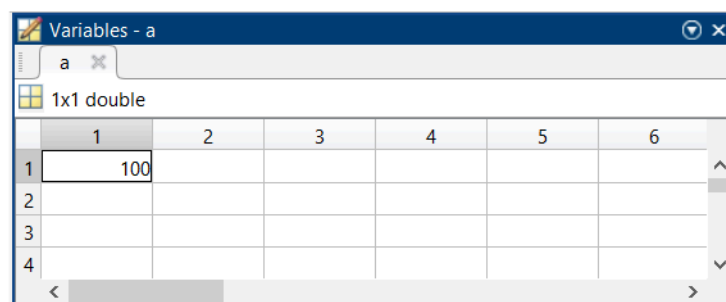


Figure 1.6: Editing a variable directly

- (l) **Confirming variable values at the command line.** If you would simply like to confirm the current value of a variable, one easy way to do this is to type the variable name at the command line, without a semi-colon afterwards. Try this now!
- (m) **Saving variables.** You can save variables in the Workspace to a MATLAB specific file format called a MAT-file using the `save` command.

To save the Workspace to a MAT-file named `tutorial_data.mat`, enter the command:

```
>> save tutorial_data
```

You should see that a file has appeared in the Current Folder pane.

- (n) **Emptying the workspace.** It is often necessary to tidy up your workspace, for example when switching to a new problem. You can remove all the variables from your workspace with the `clear` function:

```
>> clear
```

- (o) **Loading variables.** The workspace is now empty. We can load the previously saved variables by using the `load` command:

```
>> load tutorial_data
```


- (p) **Clean up the Command Window.** While the `clear` function cleans up the workspace, you can use the `clc` command to clean up the Command Window.

```
>> clc
```

---

## Exercise 1.2 MATLAB Scripts

When writing more involved MATLAB code, and particularly when there is a desire to reuse the code in the future, creating a MATLAB *script* is more appropriate than typing line-by-line at the command prompt. A script is a file that can be edited, commented, executed, and saved for future use, and it has the file extension **.m**. This exercise will demonstrate how to write a simple script.

- (a) **Create a MATLAB script.** There are a few ways to create a new MATLAB script within the MATLAB environment. The most straightforward way is to  on the **New Script** button on the MATLAB *Home* menu as shown in Figure 1.7. Alternatively, you can perform (Ctrl + N) when either the Current Folder or Command Window panes are active.

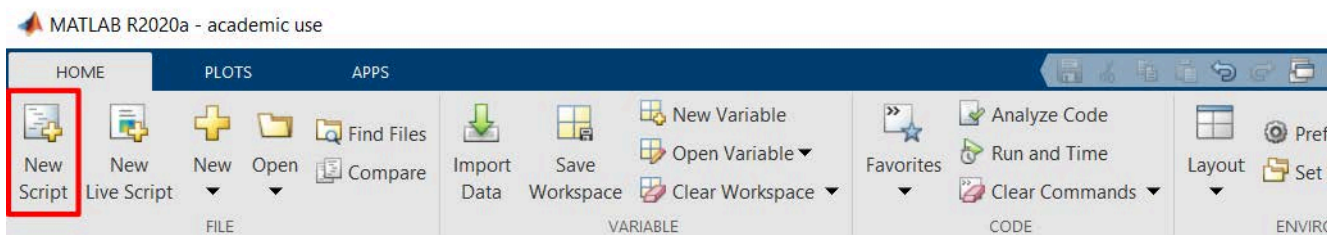



Figure 1.7: New script button on MATLAB Home menu

- (b) This will open a script window, which will initially have the name 'Untitled'. It can be saved by  on the **Save** button from the *Editor* menu, as shown in Figure 1.8, or by executing (Ctrl + S) when the script is selected.

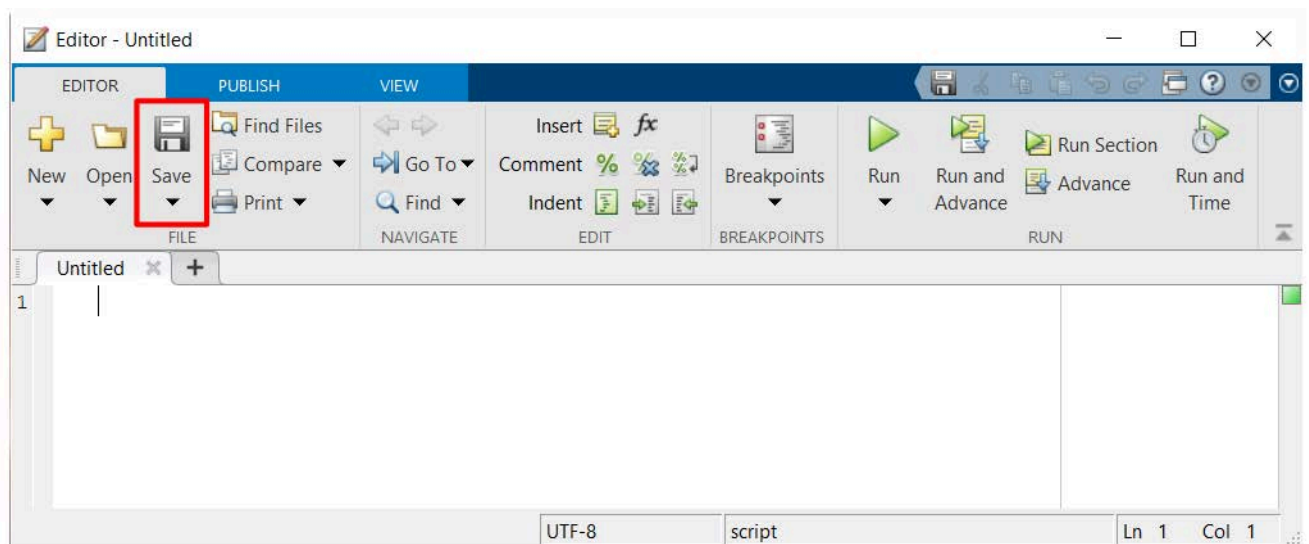




Figure 1.8: Saving a MATLAB script

Save the file as:



`\ex_1_2\my_script.m`

- (c) Change the working directory to where the script is located. This can be achieved by  on the  icon and navigating using the resulting window or using the Current Folder pane on the left of the MATLAB environment.



- (d) Next, let's enter some comments and code into the file, as shown below. Notice that comments are preceded with a `%` symbol, which turns the rest of the current line green. MATLAB will automatically recolour features of the code, in particular keywords and strings, while you type.

```
%% An Introductory MATLAB Script


a_array = [1 5 -4 9 7 -6 0];           % an array of numbers to test
a_mean = mean(a_array);

prompt = 'What is the threshold value? Enter an integer.\n';
thresh = input(prompt);                % threshold to test against

if a_mean > thresh
    fprintf('The average value exceeds the threshold.\n');
else
    fprintf('The average value is less than the threshold.\n');
end
```

Save the file once you have finished entering the code.

This simple script finds the average of the set of 7 numbers defined in `a_array`, and then compares it to a user entered threshold. One of two messages is printed to the MATLAB command line using the `fprintf` function, depending on whether the average of the array is greater than or less than the threshold. The strings all end with the newline character, `\n`, to instruct the cursor to move to the next line on completion.

- (e) **Execute the MATLAB script.** Next,  on the **Run** button to execute the script. View the MATLAB command window — there should be a message asking you to enter a threshold value. Enter a value and inspect the result, which should inform you if the array average is greater than or less than the threshold.



**Figure 1.9:** Running a MATLAB script

- (f) **What is the average value?** Although the message states whether the average is above or below the threshold, it does not give the numerical value of the average. See if you can find this out!
- (g) **Printing the results to the command line.** It may be convenient to write results at the command line using a string format. This can be done by amending the current script. Replace the if-statement at the end of the file with:

```
if a_mean > thresh
    fprintf('The average value (%f) exceeds the threshold (%d).\n', ...
        a_mean, thresh);
else
    fprintf('The average value (%f) is less than the threshold (%d).\n', ...
        a_mean, thresh);
end
```


The messages output are constructed using *conversion characters*. This allows for variables to be used within strings. The conversion character `%f` is used for floating-point numbers and `%d` is used for base 10 integers. More information on how to use conversion characters and the `fprintf` function can be found in the documentation (`>> doc fprintf`). The use of dots (`...`) at the end of the lines allows the command to be split over two lines. (This method of splitting across lines can be used for other code too, not just this particular function.)

- (h) **Execute the modified MATLAB script.** Next, save and run the modified script, and ensure that it prints the expected results to the command line.
- (i) If you wish, try changing the numerical values in the array to verify that the correct results continue to be generated.
- (j) **Write to a file.** In addition to displaying strings in the Command Window, `fprintf` can instead be used to write to files.

Update the script with the following:

```
FID = fopen('my_file.txt', 'w'); % open a file for writing
if a_mean > thresh
    fprintf(FID, 'The average value (%f) exceeds the threshold (%d).\n',...
        a_mean, thresh);
else
    fprintf(FID, 'The average value (%f) is less than the threshold (%d).\n',...
        a_mean, thresh);
end
fclose(FID);
```

We can create a file for writing to by using the `fopen` function. In this instance, we have created a file called `my_file.txt`. The second argument is the permission granted when interacting with the file. Here, `'w'` opens the file for writing, discarding the existing contents. You can inspect other permissions by reading the documentation (`>> doc fopen`). By using the file identifier, `FID`, as the first argument of `fprintf`, the string is instead written to the file associated with the identifier as opposed to being displayed in the Command Window.


Re-run the script and inspect the Current Folder pane. A new text file should have appeared.  on the name to open it, and confirm that it contains the expected string.

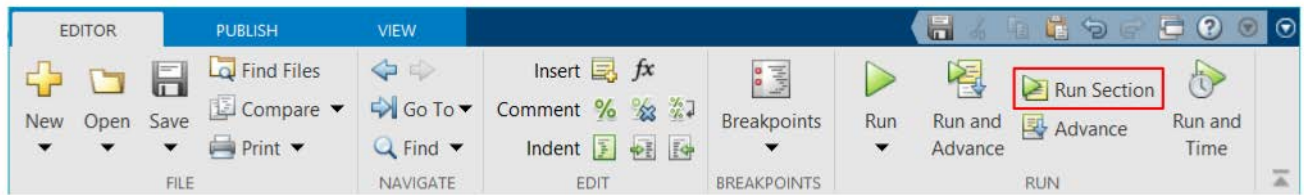
- (k) **Read from a file.** Strings can be read from a file and assigned a variable. Append the following to the script.

```
%% Read from the file

txt = fileread('my_file.txt');
disp(txt);
```

You may have noticed that a horizontal line has appeared across the script, above the comment prefaced with two percentage signs (`%%`). This is because, in MATLAB syntax, two percentage signs at the start of a line indicate a new section and the text which follows is referred to as a *section title*. This is useful as often you may wish to focus your attention on specific aspects of your code and run these independently from other sections.

- (l) **Run the section.** To run a section, first click on the section you wish to run (the active section will be highlighted in yellow) and then  on the *Run Section* button in the *Editor* menu, as shown in Figure 1.10. Alternatively, execute (Ctrl + Enter) after selecting the section to be run.



**Figure 1.10:** Running a MATLAB script section

A variable `txt` will have appeared in the Workspace which contains the contents of the file, `my_file.txt`. Another function, `disp`, has output the variable to the Command Window.

---

## 1.2 MATLAB Arrays and Matrices

MATLAB is a tool for technical computing, and it has extensive support for working with data in the form of arrays, matrices, and structures. In fact, MATLAB is an abbreviation for MATrix LABoratory! Arrays and matrices are crucial for many types of mathematical analysis.

The next few exercises will introduce these data types and demonstrate some aspects of their use. There are many more functions relating to arrays and matrices (in particular) than can be covered here; however, following the examples in the coming exercises will provide a good start!

---

### Exercise 1.3 MATLAB Arrays

Earlier exercises have introduced MATLAB variables which have been scalars. In this exercise, we will work with arrays, and highlight some useful functions for working with arrays.

The following steps can either be executed at the MATLAB command prompt, or in a MATLAB script, as you prefer. Remember that you can type ‘doc’ followed by the name of the function, to open the help documentation on that topic.

- (a) **Define arrays.** An array of values can be specified easily in MATLAB. Enter the following code to create two new arrays.

```
my_array_r = [1 -2 -5 6, 9, -6, -4 0]    % define a row vector
my_array_c = [8; -9; -1; -4; 3; 6; 5]    % define a column vector
```

Notice that when creating a row vector, either a space or a comma can be used to separate values. Column values must be separated using a semi-colon.

- (b) **Create an array of zeros or ones.** Sometimes it is useful to create an array of a certain dimension, where all of the elements are zero (0), or all elements are one (1). This can be easily achieved via the following code:

```
my_ones = ones(1,10)    % create an array of ones, with 1 row
                        % and 10 columns
my_zeros = zeros(1,8)    % create an array of zeros, with 1 row
                        % and 8 columns
```

- (c) **Find the length of an array.** To find the length of a previously declared array, type:

```
num_ones = length(my_ones)    % find the length of array my_ones
num_zeros = length(my_zeros)    % find the length of array my_zeros
```

It can be useful to find the size of an array when defining the limit of a ‘for’ loop, for example (the ‘for’ loop might be required to iterate once for every element in the array).

- (d) **Find the maximum or minimum value within an array.** Try this code to find the maximum and minimum values of the arrays defined earlier.

```
max_r = max(my_array_r)      % find the largest value in my_array_r
min_c = min(my_array_c)      % find the smallest value in my_array_c
```

You can also combine the above with the `abs()` function, if you would like to find the maximum and minimum absolute values.

- (e) **Generate an array of random values.** To create an array of random numbers, type:

```
my_rands = rand(1,10)        % create an array of random values
```

This will create 10 random values from a uniform distribution, in the interval 0 to 1. If you would like to generate random values across a different range, then you can do this by scaling and/or offsetting the generated array. For instance, if we wanted to create random values in the range -6 to +6, it could be done like this:

```
my_rands_6 = (12*rand(1,10))-6    % create randoms in range -6 to +6
```

- (f) **Obtaining integer values.** If the desired random array is intended to contain integer numbers only, the `round()` function can be incorporated to achieve this:

```
my_rand_ints = round((12*rand(1,10))-6)    % create random integers
```

- (g) **Transpose an array.** Arrays can be transposed in a couple of different ways. Try these out, to transpose the `my_rands` and `my_rands_6` arrays (in this case, transposing will change both arrays from a single row to a single column).

```
my_rands_t = transpose(my_rands)    % transpose the first array
my_rands_6_t = my_rands_6'          % transpose the second array
```

- (h) **Extract a specific element from an array.** If you would like to find the value of a particular element from an array, this can be done simply by giving the desired index in brackets. It is important to note that MATLAB indexing begins at 1 (rather than 0, as in some other programming languages).

```
first_elem = my_rands(1)            % pulls out first element
zero_elem = my_rands(0)             % try this, and confirm it fails!
```

- (i) **Find particular elements in an array.** Sometimes it is useful to extract specific elements from an array, and the first step in doing so is to identify the indices of these elements. We might do so by testing which

elements of an array meet a desired condition. For instance, suppose that we would like to find those elements of the array `my_rands_6` that are less than zero.

```
sub_zero_i = find(my_rands_6 < 0)      % find indices of elements < 0
my_rands_6_sz = my_rands_6(sub_zero_i) % extract sub_zero elements
```

Note that `sub_zero_elems` is an array of the *indices* of `my_rands_6` that meet the condition, NOT the values themselves. The second operation generates a new array of values that contains only those elements of `my_rands_6` (the original array) that meet the condition.

- (j) How would you concatenate the arrays `my_ones` and `my_zeros`, to form a single, longer array? Use the Help documentation to find out.
- (k) There are many more possibilities for working with arrays than we can cover here — to investigate further, please refer to the Help documentation:

```
doc matrices and arrays
```

---

---

**Exercise 1.4 Matrices in MATLAB**

Matrices are a natural extension of arrays, comprising two dimensions of data (or higher dimensions in some cases), and we can define and operate on matrices easily in MATLAB. This exercise introduces a few common operations, based on two dimensional matrices.

- (a) **Define a matrix.** Matrices can be defined in a very similar manner to arrays. For instance, to define two matrices of values with 3 rows and 4 columns, we can simply write:

```
my_matrix_A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
my_matrix_B = [3 3 3 3; 1 2 1 2; 4 4 5 5]
```

- (b) **Find the dimensions of a matrix.** Similar to the `length()` function which we applied to arrays, the `size()` returns the dimensions of a matrix, giving first the number of rows, and then the number of columns. Try checking the dimensions of `my_matrix_B` by typing:

```
size(my_matrix_B) % find the dimensions of the matrix
```

- (c) **Create matrices containing zeros, ones, or random values.** These three types of matrices can be created using the same syntax as shown in Exercise 1.3 for arrays — the difference is now that both arguments supplied to each function are greater than 1, because there is no singleton dimension in a matrix, as there is in an array.

```
my_oneM = ones(4,2) % create a matrix of ones, with 4 rows
                    % and 2 columns
my_zeroM = zeros(3,5) % create a matrix of zeros, with 3 rows
                    % and 5 columns
my_randM = rand(5,5) % create a matrix of random values, with
                    % 5 rows and 5 columns
```

- (d) **Matrix addition and subtraction.** Matrices can be added and subtracted, meaning that these operations are performed on individual elements, provided that the dimensions of the two matrices are equal. For instance, try typing:

```
my_matrix_C = my_matrix_A + my_matrix_B % add matrices
my_matrix_D = my_matrix_A - my_matrix_B % subtract matrices
```

- (e) **Element-wise matrix multiplication.** Following on from last point, matrices can also be multiplied on an element-wise basis, using the `.*` operator, again provided that they have the same dimensions. Try this now, using `my_matrix_A` and `my_matrix_B` as the multiplicands.
- (f) **Matrix product.** Matrices may be multiplied, provided that the dimensions agree. This means that the number of columns in the first matrix, and the number of rows in the second matrix, must be the same. It would not be possible to multiply the matrices `my_matrix_A` and `my_matrix_B` defined above, for

instance, because their dimensions are not compatible (if you wish, you can try this and see what happens!).

Try defining matrices that *will* be compatible for multiplication, and find the matrix product (using the `*` operator).

- (g) **Extracting a row or column from a matrix.** It is useful to review how to extract a row or column from a matrix, to form an array. This can be useful, for instance, when a matrix represents several channels of data, and we require to separate them. To illustrate retrieval of a row and column, we can extract the first row, and then the second column, of the matrix `my_matrix_A`.

```
row_1 = my_matrix_A(1,:)           % extract first row
col_2 = my_matrix_A(:,2)           % extract second column
```

- (h) **Transpose a matrix.** The transpose of a matrix can be found using the same method as the transpose of an array. Refer back to Exercise 1.3, and adapt this code to find the transpose of `my_matrix_A`.
- (i) **Squeezing a matrix.** Occasionally, variables can be created with more dimensions than are necessary. There is effectively a superfluous dimension. This extra dimension can be removed, and the data converted to a  $1 \times n$  array, by applying the `squeeze()` function.

First, let's create a variable with an 'extra' dimension...

```
my_var(1,1,:) = [1 2 3 4 5 6 7 8 9]    % create 1 x 1 x n data
```

You can confirm these dimensions by looking in the MATLAB Workspace.

Next, we can 'squeeze' the variable to remove the unnecessary dimension:

```
my_var_2 = squeeze(my_var)              % reduce to 1 x n data
```

- (j) As noted previously, there are many more possibilities for working with matrices than we can cover here — to investigate further, please refer to the Help documentation, by typing:

```
doc matrices and arrays
```



### 1.3 Plotting in MATLAB

Our next example in this brief introduction to MATLAB demonstrates how to create basic figures. As with many other aspects of MATLAB, there are far more possibilities than can be covered here, but it is useful to provide a few simple examples to help new users get started.

#### Exercise 1.5 MATLAB Figures

This exercise will demonstrate how to plot sine and cosine waves, add axes labels, legends and titles, and demonstrate how to customise different aspects of their appearance.

- (a) **Create a new script.** Open a new MATLAB script and save it in your working directory, with the name 'plot\_sin\_cos.m'. You may wish to create a new directory for this exercise.
- (b) **Defining header information and parameters.** First, add a line or two of comment to explain that this file will be used to generate and plot sine and cosine waves. Then, enter the following lines of code to specify the sampling frequency to be used, the frequency of the sine and cosine waves, and the duration of the simulation.

```
% script to generate and plot sine and cosine waves

fs = 1000;           % sampling frequency in Hz
f1 = 100;            % frequency of sine and cosine waves in Hz
Tmax = 0.1;          % duration of simulation (stop time in seconds)
```

- (c) What is the sample period? Add a line of code to calculate the sample period, and assign it to the variable Ts.
- (d) **Generate the data arrays.** The next step is to generate an array of time samples extending from 0 to Tmax = 0.1 seconds, which we will call t. The corresponding sine and cosine values can be generated according to equations:

$$A_s = \sin(2\pi f_1 t) \quad (1.1)$$

$$A_c = \cos(2\pi f_1 t) \quad (1.2)$$

... which will result in another two arrays for sine and cosine respectively. Add the following code to your MATLAB script to create the time, sine and cosine arrays.

```
t = [0:Ts:Tmax];      % create an array of time values from 0 to Tmax
                        % (in steps of Ts, the sample period)
As = sin(2*pi*f1*t);   % create a corresponding array of sine samples
Ac = cos(2*pi*f1*t);   % create a corresponding array of cosine samples
```

- (e) **Run a preliminary simulation.** Often it is useful to run a script from time to time while you are writing it, so that you can detect any problems at an early stage. Run the script now and check that there are no errors (which would be shown in red in the command window), and that the variables have been created

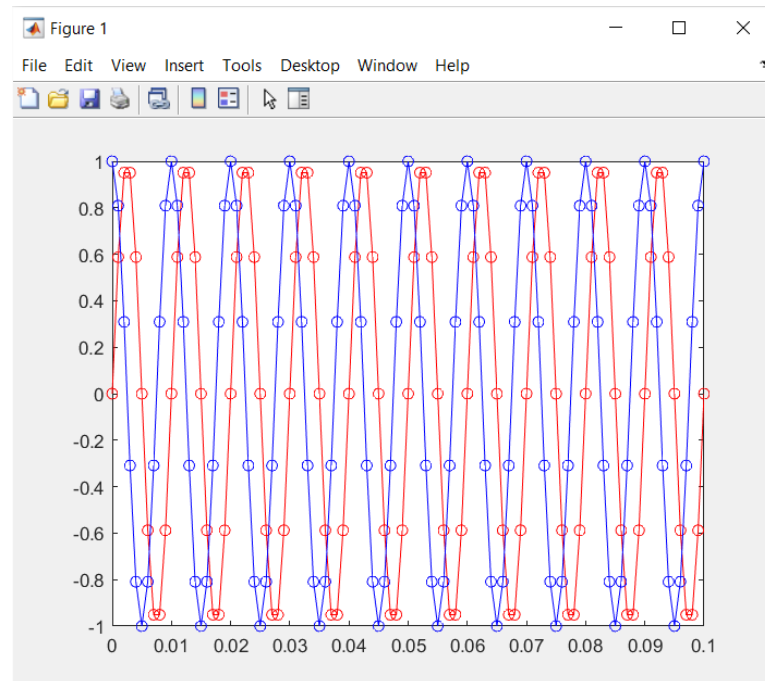
successfully. If any problems have cropped up, tend to these now before moving on! Note that the script will automatically be saved when you execute it.

- (f) **Create a figure and plot data.** Next, we must add some lines of code to generate a new figure (it will be numbered as Figure 1), and plot the sine and cosine data within it. Copy the lines below into your script (the comments are optional!).

```
figure(1)           % create a new figure
hold off           % do not retain any previous data in plot
plot(t,As,'r-o');   % plot the sine wave data in red ('r') with
                   % a continuous line and a round marker

hold on            % retain the sine while we add the cosine...
plot(t,Ac,'b-o');   % plot the cosine wave data in blue ('b')
                   % with a continuous line and a round marker
```

- (g) **Run another preliminary simulation.** Re-simulate to inspect the results! You should see a figure window appear as in Figure 1.11.



**Figure 1.11:** Plot of sinusoid and cosine waveforms

- (h) Compare the appearance of the figure against the code written so far. It should be possible to see that the cosine appears in blue and the sine in red, as desired. Both are continuous lines with round markers, as a result of the 'b-o' and 'r-o' strings supplied as part of the `plot()` function calls.
- (i) **What does 'hold on' do?** Try commenting out:

```
hold on
```

... by adding a `%` symbol to the start of the line, and then re-simulating. You should notice that only the cosine wave is shown now — the sine wave has not been retained when the next `plot()` function is called. The purpose of the hold is to enable multiple sets of data to be plotted on the same figure.

- (j) **Experiment with line and marker styles.** Try changing the specification of the two lines and markers to alter their appearance. For instance, you could test the effect of changing the specifier string to 'm:s' or 'k--x'.

Further possibilities include thickening the line, changing the size of the marker, or filling the marker (if it is hollow — a square or circle, for instance — other, non-fillable marker styles include crosses and asterisks). The fill colour of the marker can also be specified, and it need not match that of the line.

Try changing the code so that the two lines starting with `plot...` are as follows, then re-simulate and see the effect of your changes.

```
plot(t,As,'b--s','MarkerFaceColor','m','LineWidth',1,'MarkerSize',5);
plot(t,Ac,'k-o','MarkerFaceColor','y','LineWidth',2,'MarkerSize',10);
```

You might also wish to consult the MATLAB Help documentation to find out more about the possibilities of the `plot()` function!

Once finished, return the line and marker styles to their original specifications.

- (k) **Label the axes, and title the plot.** Although data has been drawn on the figure, it does not yet have any axes labels, or a title. You can add these now by appending the following lines of code to your script.

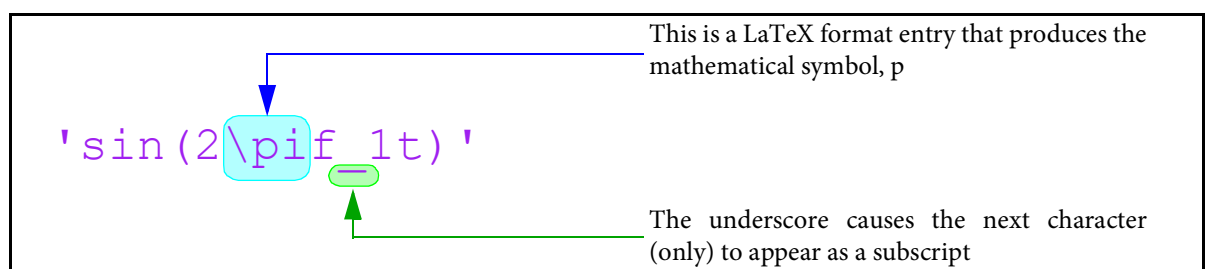
```
xlabel('Time (seconds)');      % add a label to the x (time) axis
ylabel('Amplitude');          % add a label to the y axis
title('Sine and Cosine');      % add a title for the plot
```

- (l) Once incorporated, re-run the simulation to observe the changes. You should now see that the labels and title specified in the code have been added to the figure.
- (m) **Add a legend and customise formatting.** The last step we will take is to add a legend to the figure, to clearly indicate which set of data represents the sine wave, and which represents the cosine wave. It would also be nice to improve the appearance by changing the axis limits, and showing a grid in the background. These three features can be introduced by the code shown below. Add this into your MATLAB script.

```
legend('sin(2\pif_1t)','cos(2\pif_1t)','location','NorthEast');
grid on;
axis([0 max(t) -1.2 1.2]);
```

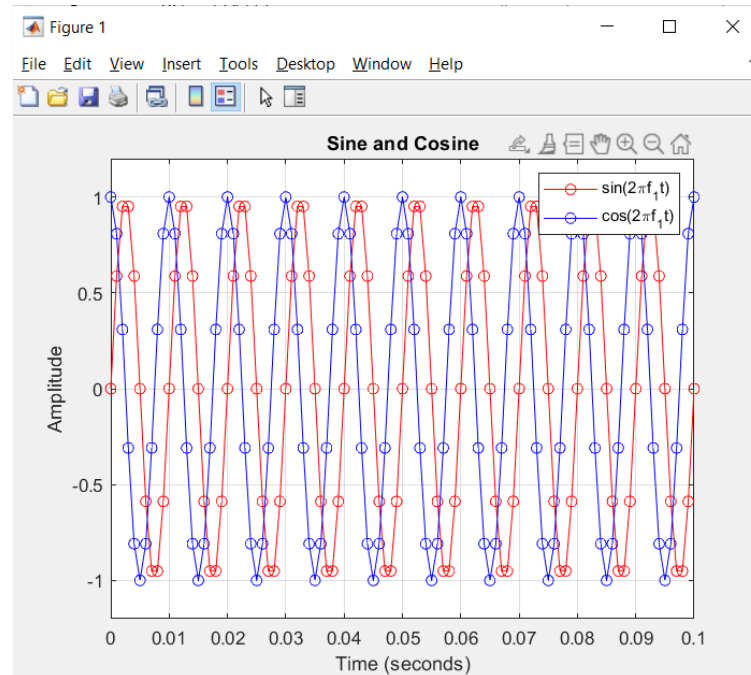
% add a legend to the graph  
% show the grid  
% configure the axis scaling

It is useful to explain a couple of points relating to the legend text, taking the first legend entry as an example.



If you wish, you can also experiment with changing the position of the legend (acceptable entries include all of the major and minor compass points, e.g. 'North', 'South', 'SouthWest', etc., and also outside the axes, e.g. 'EastOutside', 'SouthOutside').

- (n) Once you have added the extra code, re-run the simulation to see the effect. The legend should now be visible, along with the grid. Notice also that the axis limits have been changed to provide a little extra space above and below the sine and cosine waves. Your final version of the figure should look similar to this:



**Figure 1.12:** Plot of sinusoid and cosine waveforms with all the trimmings

- (o) **Challenges!** Now that you have followed through this example to produce the desired graph, consider the following points, and see if you can alter and simulate the script accordingly.

Can you change the duration of the simulation to 0.6 seconds, ensuring that the time axis is altered appropriately?

See if you can alter the sampling frequency such that the signal is sampled at 2kHz.

Can you change the amplitude of the sine wave to 0.7, and the amplitude of the cosine wave to 1.3? Make sure that the figure shows the full range of both.

Try adding a third data series to the figure. Add the sine and cosine together, and plot this in magenta with square markers. Make sure that your extra plot is also labelled.

What happens if you supply only the sine and cosine wave samples to the `plot()` function, without the time information?

Check out the `stem()` function... Replace `plot()` with `stem()` and see what happens!

## 1.4 MATLAB Functions

There are many functions available in MATLAB, including simple arithmetic calculations like `rem()`, and string formatting functions such as `num2str()`, which was demonstrated in Exercise 1.2. Functions allow frequently used pieces of code to be packaged for easy reuse. For instance, we often want to find the average of a set of numbers — one method would be to add all the numbers up, and then divide by the number of numbers(!), but actually it is much easier to simply use the `mean()` function.

There are also a wide variety of other functions available in MATLAB and its associated toolboxes. The particular selection of functions available to you will depend on the products you have installed on your computer, e.g. if you have the Signal Processing Toolbox installed, you will be able to use filter design functions such as `fir1()`.

In the next couple of exercises, we will further investigate built-in functions of MATLAB and its toolboxes, and then go on to consider the creation of custom functions.

---


### Exercise 1.6 Functions in MATLAB



In this exercise, we demonstrate how to investigate the functions available in MATLAB and its toolboxes. As you will see, there are too many functions to introduce them all, but that is exactly what the Help documentation is for!

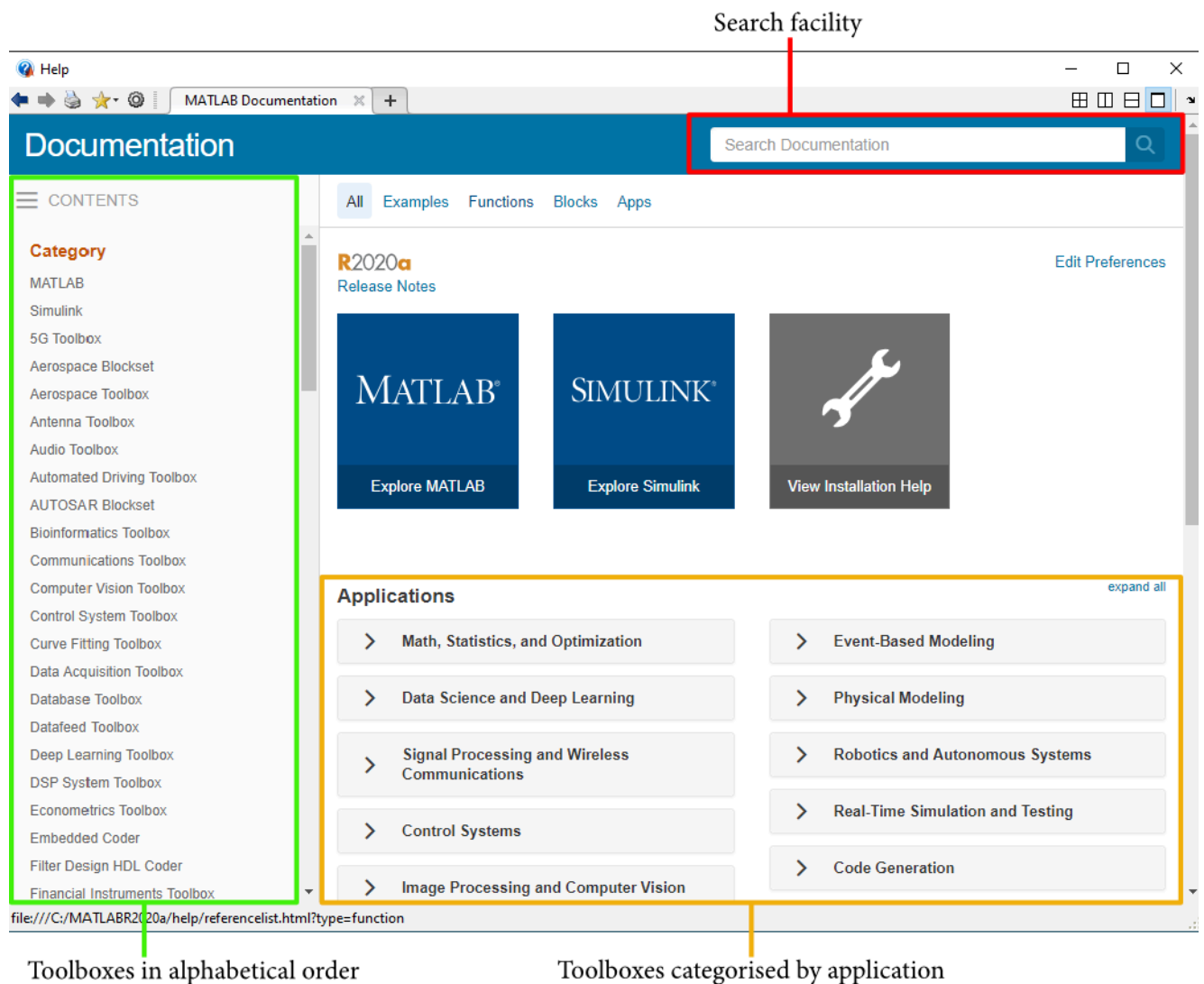
- (a) **Obtain information about functions.** At the MATLAB command prompt, type

```
>> doc
```

and then *Enter* to open the Help browser. You should now see a window, similar to Figure 1.13, listing all of your installed components.

If you are looking for a function to perform a certain task (e.g. for a DSP or communications application), then  on the drop-down named *Signal Processing and Wireless Communications*. Alternatively, you can search using the box at the top of the window.

- (b) Let's assume that we would like to find out about the `fir1()` function mentioned earlier. This relates to digital filtering, so we will look in the Signal Processing Toolbox for more information.  on the link to the *Signal Processing Toolbox*.
- (c) At this point, you should see the contents of the Help on the Signal Processing Toolbox. It is arranged based on different types of signal processing operations (and you may wish to explore some of these!). You should also notice that there is a **Functions** link at the top of the page —  on it.
- (d) The view should now change, to show the selection of signal processing functions available. Again, these are listed according to category. You may wish to take a few moments to review the available functions.
- (e) Locate the `fir1()` function in the list, and select it, to see more information about this function.
- (f) Read through the information, noting that `fir1()` can be called using different sets of arguments (inputs to the function). The *Description* section provides detailed information about the operation of the function, and towards the bottom of the page, some *Examples* are provided to demonstrate its use.
- (g) **Replicate an example.** MATLAB Help often provides segments of example code. These can be cut and pasted into the MATLAB command window, and run, to replicate the results shown in the documentation. Try this now for Examples 1 and 2, and check that the results correspond with the Help file.



**Figure 1.13:** MATLAB documentation home page

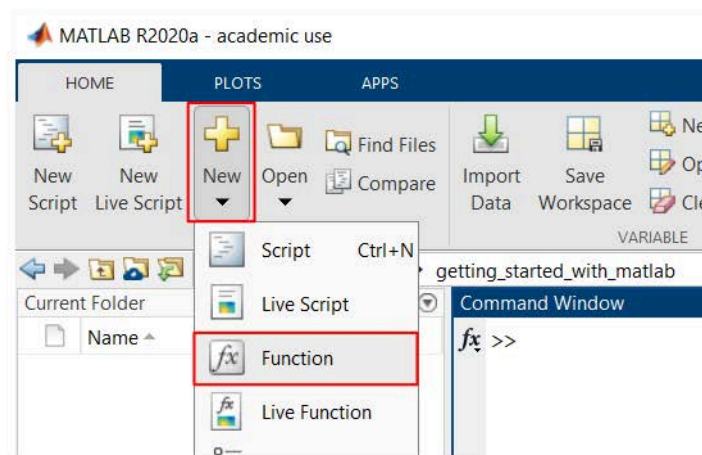
- (h) What other functions are used within these example code segments? Can you use the documentation to find out more about them?

## Exercise 1.7 Writing Your Own Functions

Although MATLAB contains many functions, in some cases you may wish to write your own, to undertake a custom operation. Next, we will demonstrate how to write a new function based on the averaging-and-thresholding script from Exercise 1.2.

Bear in mind that our example is a simple one, and functions can be written using many combinations of input and output types, options, etc. They can also be ‘overloaded’, such that the function behaves differently depending on the supplied set of input arguments. We do not have time to cover these more complex examples here, but it is useful to be aware that you can create your own, sophisticated functions in MATLAB — see the Help documentation for more information.

- (a) **Create a new function.** In the main MATLAB window, choose the *New Function* option as shown in Figure 1.14 below.



**Figure 1.14:** Creating a new function from the toolbar

A new editor window will now open, containing a function template. The initial name of the file is ‘Untitled’ (or similar), and the name of the function is also ‘Untitled’. **It is important to note that the name of a function, and its file name, must be identical.**

- (b) Save the function with the name **compare\_mean\_to\_thresh.m**.
- (c) **Customise the function interface.** Change the function name from `Untitled` to `compare_mean_to_thresh`, and replace the `input_args` placeholders with the argument names `num_array` and `thresh`, separated by a comma. Similarly, change `output_args` to a single argument called `diff`.
- (d) When called, the function will be passed an array of numbers, `num_array`, and a threshold value, `thresh`, as the input arguments. It will then calculate the difference between the average of the array values, and the threshold, and return the result as the output argument, `diff`. The value of `diff` should be positive if the mean is higher than the threshold.

Write some comments to explain the operation of the function, replacing the placeholder comments in the template.

- (e) **Write the function!** Next, write your own MATLAB code to implement the operation of the function described above, and then save the file.

- (f) **Call the function.** We can now call the function from other MATLAB code, provided that the file `'compare_mean_to_thresh.m'` is in the current working directory (or its location has been added to the MATLAB path). The function can be called from a MATLAB script, or directly from the command line.

Type the following commands (either into a script, or individually at the command line), and execute them. You do not need to enter comments if typing at the command line.

```
% code to demonstrate a function call

clear all;                                % clears Workspace of existing vars
nums = [3 10 7 -8 -2 5 -6 -1 4];          % an array of numbers to test
thresh = 2;                               % threshold to test against

% CALL THE FUNCTION!
% (omit semi-colon at end of line to print result in command window)
difference = compare_mean_to_thresh(nums,thresh)
```

Check that the function executes successfully and produces the correct result!

- (g) **Further function investigation!** Here, we have taken the example of a function that returns a single result. It should be noted, however, that functions can recall more than one result, or indeed they can return no results! Find out more about this topic using the Help documentation, by typing

```
doc function basics
```

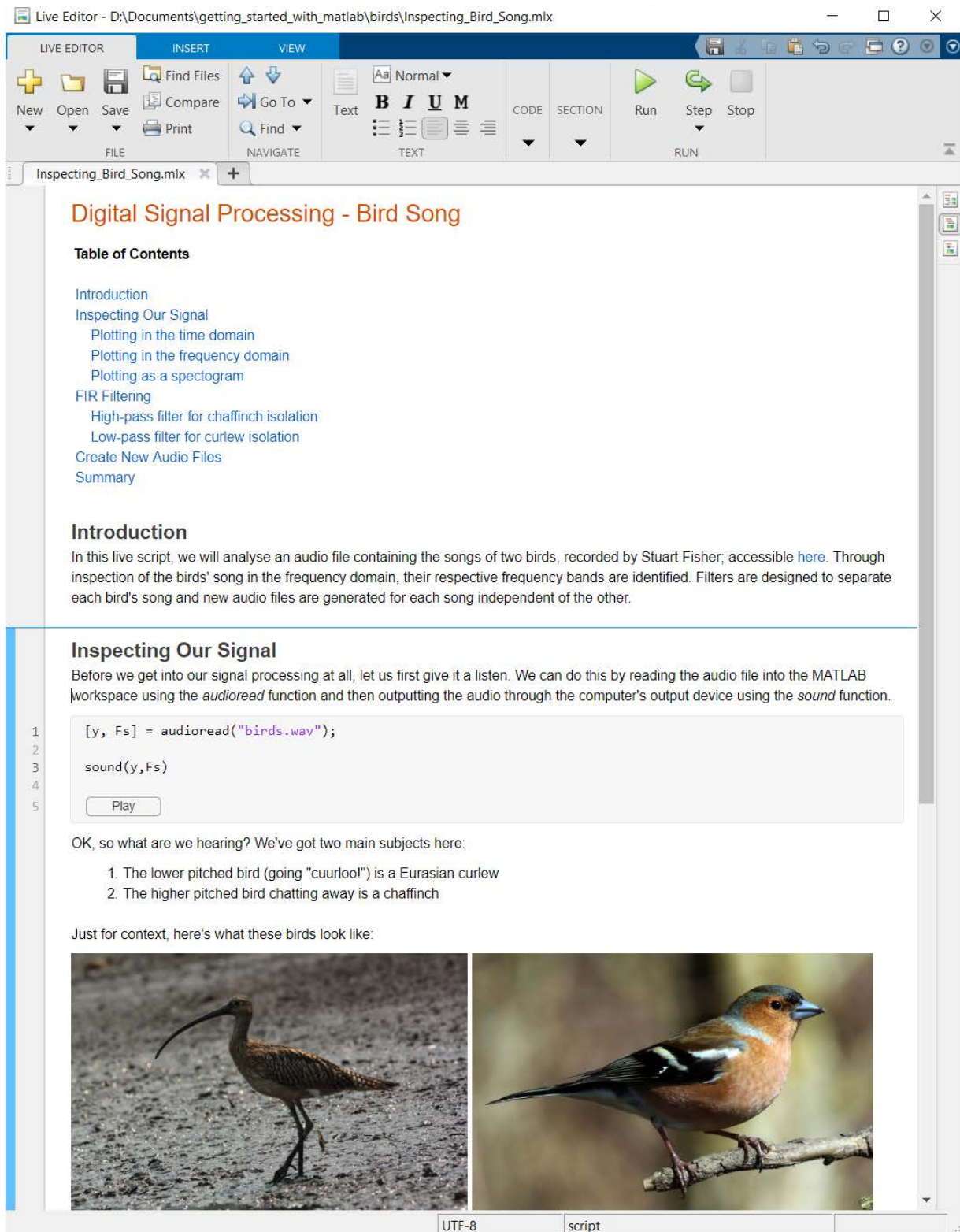
at the command prompt.

---



## 1.5 Live Scripts

While scripts were introduced earlier, in Exercise 1.2, MATLAB also provides a file format called *Live Scripts* which have the extension **.mlx**. These differ from regular scripts in that they are interactive and can contain more than code alone. In addition to MATLAB code, live scripts can contain text, headings, images, plots, equations, control widgets, and more! This can be very useful when making work reproducible or conveying ideas as a narrative.



**Figure 1.15:** An example of a MATLAB live script

---

**Exercise 1.8 Digital Signal Processing of an Audio File**

This final exercise is more of a project where the goal is to separate the song of two birds from one another. An audio file has been provided which contains a recording of two birds (curlew and chaffinch) signing at the same time. You should create a live script that introduces the problem and follows a procedure for isolating each song.

An example template may take the form:

- (a) Import the \*.wav file into the MATLAB workspace.
- (b) Inspect the signal in the time and frequency domain using plotting.
- (c) Design digital filters using parameters ascertained from frequency inspection.
- (d) Perform filtering.
- (e) Generate new audio files.

Some useful functions:

- Audio related functions: *audioread*, *sound*, *audiowrite*
- Frequency domain functions: *fft*
- Plotting functions: *plot*, *spectrogram*
- Filtering functions: *highpass*, *lowpass*

Remember to use:

`>> doc function_name`

to find out more information about functions.

---

