3
Elements of ... programming

NERD. This is what you are now going to become. And lose all your social skills. And sit at home all day in front of your computer. Which has become your only friend.

You will achieve this higher state of Being by starting to learn to write and use *scripts* and *functions* (aka `m-files`) in **MATLAB**. Actually, at this point you are now writing computer programs (of a sort) rather than endlessly typing stuff at the command line in the forlorn hope that something useful might occur. You will also be doing a great deal of code debugging ...

3.1 Introduction to scripting (programming!) in MATLAB

Commands in MATLAB can become very lengthy, and you typically end up with multiple lines of code to get anything even remotely useful done. And as you have noticed, it can take a lot of time to enter in all these lines. When when you log off and go home ... it is all gone. [1] ... If only there was some way of storing all these commands in such a way that they could be worked on and run again with the press of a button (as a wild guess, how about **F5**?), without having to enter them all in, all over again from scratch ...

Your wish is granted. In **MATLAB**, it is possible to store all of your commands in a single text file, and then request that they are all executed (sequentially) at one go. **MATLAB** gives this text file a fancy name (because it is a very fancy piece of software, after all) – a *script*[2], otherwise known as an **m-file**. To create a new `m-file`; from the File menu, select **Script** (a common type of **m-file**)[3]. You will see a text editor (more fancy-ness) appear in front of your very eyes, containing your requested (but currently empty) **m-file**. Save the **m-file** to your directory of choice. Alternatively, simply create a new (blank) text file and saving it with the extension `.m`, rather than e.g. **.txt**, creates you a (script) **m-file**. From an **m-file**, you can issue all the **MATLAB** commands you previously would have entered individually, line-by-tedious-line, at the command line. Furthermore, having created and saved a **MATLAB** script, it can be executed again

[1] **MATLAB** remembers all the commands used in previous session (although this may not be the case of shared, lab computers) and lists them in the **Command History window**. You can recover and re-execute a previous command in this list by double-clicking it. You can also re-run more than one line at a time by selecting multiple lines and pressing **F9** (or Evaluate Selection from the (R-mouse button in Windows) context menu).

`m-file`
... is nothing more than a simple text file, in which a series of one or more **MATLAB** commands are written and which via the **.m** extension, **MATLAB** interprets as a program file (*script* or *function*) that can be edited and executed (or rather, the list of commands inside, can be executed in sequential order).
Assume a similar convention to that for *variables* in the naming of `m-files`.

[2] The conception of a *function*, will be introduced later.

[3] In order version of **MATLAB**: **File**/**New** menu, and select: **Blank M-file**.

and as many times as you like.

You can execute an **m-file** by typing its name into the **Command window** (omitting the **.m** file extension). Ensure that **MATLAB** is operating in the same directory as the directory that you have saved your **m-file**. You can also run the *script* (**m-file**) by hitting the big bright green Run icon button at the top of the **m-file** editor[4]. The short-cut for running it is to whack your paw down on the Function Key **F5**.

OK – you are now ready for your very first program ... inevitably ... this has to be to print 'Hello World' to the screen. No, really. (Google it.) Create a new **m-file**, calling it e.g. `hello_world.m`. You need to use the function `disp` (see Box or type » `help disp`) as always, for function syntax and usage), which will print to the screen, either any text you specify (in inverted commas), or the value of a variable (which could also contain character information). For now, simply pass the text directly. Your program needs just a single line in the **m-file**:

```
disp('hello, world')
```

Save the file (to your working directory). Run it at the command line by typing its name (omitting the **.m** extension). Your first program is a success! (Surely you could not screw up a single line program ... ?[5]) You could extend this to a mighty 2-line program by defining the string as a variable and displaying the contents of the variable, i.e.,

```
message = 'hello, world';
disp(message)
```

For further practice – pick one of any of the previous exercises in which multiple lines of code were required, place them into a new **m-file** (either by re-typing them in or copying them out of the **Command History window**), save the file (to the same directory that you are working from), and run it my typing its name at the command line (omitting the **.m** extension).

3.1.1 *Programming good practice*

A few tips about good practice in (**MATLAB**) programming before we go on (and on and on and on):

- Choose helpful variable names so that it is clear what each variable represents. Avoid *excessively* short names, except for simple index and counting variables. At the other extreme – excessively long names, which the might be wonderfully descriptive, can lead to even simple calculation stretching over multiple lines of code (which can make it more difficult to see what is going on in the code overall).

[4] In order versions of **MATLAB** – select: **Debug/Run** from the 'debug' menu of the **Editor window**.

[5] If **MATLAB** gives you an error message something like
`Undefined function or variable 'hello_world'`
then it is likely you are simply not in the same directory as the **m-file**, and/or the location of the **m-file** is not in one of the directory paths **MATLAB** knows about (see previous Tutorials for comments on changing directory vs. adding paths.).

`disp`
... displays something (the contents of a variable) to the screen. Actually, it effect is basically identical to leaving off the semi-colon (`;`) from the end of a line. In the example of:

```
disp(X)
```

where the contents of `X` is a string, you get the text displayed.

Note that the difference between using `disp` and simply typing the variable name:

```
disp(X)
```

is ... well, find out for yourself!

Creating `help` text in an `m-file`
MATLAB allows you to crete a 'help' section in the **m-file** – text that is outputted too the screen if you type help on that particular *script* (or *function*). The text is defined by a block of comment lines at the very top of the script file (or after the function definition in the case of a function). The last sequential comment line is taken to be the end of the help section. Note that the help section can be a minimum of eon single line. A typical basic format is:

1. Name of (in capitals), and very brief summary, of the script (/function).
2. List and description of the different forms of use (if there are one or more optional parameters) including definition of the input parameters.
3. Examples.
4. A `See also` section listing similar or related scripts or functions.

- Use comments within your `m-file` to add explanation and commentary on your program. Anything after a `%` on the same line is a considered a comment[6], and is ignored by **MATLAB**.
- Structure the code nicely. You can break the code up into sections, e.g. by adding a blank line. You might also start each section with a label summarizing that it is going to do (via the addition of a *comment*).
- To start with – program in as a simple step-by-step way as possible. Breaking a complex calculation into several lines of simpler calculations is much easier to debug and work out what you were doing later, particularly if comments are also added. For all practical purposes – at this level, everything will run just as fast whether as a complex calculation on one line, or simple bite-sized calculation spread over 4 lines with comment sin between.
- Always save your changes before running your program (or you may unknowingly be running the previous version).
- If using the script to do some plotting, sometimes (but not always) it is convenient to add at the top of the `m-file`,

```
close all;
```

This command close all currently open figures, plots, images, etc.

[6] Your `% comment` can start on a new line, or follow on from the end of a line of code, whichever is more helpful.

An illustration (and a far from perfect illustration) of a short script exhibiting at least a few examples of good practice, is:

```
function [dum_temp] = ch4_ebm_basic(dum_S0)
% 0D case of EBM - analytical solution
% function takes one parameter - the solar constant (units of
W m-2) [NB. modern value:  1370.0]
% define constants
const_0C = 273.15; % (units:  K)
const_sigma = 5.67E-8; % Stefan-Boltzmann constant (units:  W
m-2 K-1)
% define model parameters
par_emiss = 0.62; % (non-dimensional)
par_albedo = 0.3; % mean albedo
% solve for surface temperature
% equilibrium equation:
% (1.0-par_albedo)*(par_S0/4.0) = par_emiss*const_sigma*loc_temp^4.0
% then re-arranged to:
loc_temp = ...
( (1.0-par_albedo)*(dum_S0/4.0)/par_emiss/const_sigma )^0.25;
% convert temperature units (Kelvin to Celsius) and set value
of return variable
dum_temp = loc_temp - const_0C;
end
```

which also illustrates one possibility for variable naming convention ('constants' (variables which never change in value) start with a

`const_` and parameters (variables whose values might be changed) with `par_`, temporary ('local') variables with `loc_` and variables passed into and out of the function: `dum_`). Note use of the semi-colon at the end of every line to prevent (here unwanted) printing of results to the screen. In the file, you can create as much 'ASCII art' as you like if it helps to make the code clearer, e.g. adding separator comment lines ...

```
% ------------------------------
```

... or highlighting certain section headers, e.g.

```
% *** PLOTTING SECTION ***
```

If it (a line) starts with a percentage symbol, then **MATLAB** ignores it and you can type whatever you like after it (on the same line).

Your Hello World program might look like the following once it has had a little tune-up (although in this example this is pretty much over-kill):

```
% program to print 'Hello World' to the screen
% *** START ***
% first - define the text to display and assign it to the
variable message
message = 'hello, world';
% second - display the contents of variable message
disp(message)
% *** END ***
```

Finally, and related to the next subsection – code in stages, testing the (partial) code at each step. Do not try and write all the code in one go and only try it out at the end[7].

[7] Because it will not work 99 times out of 100 ...

3.1.2 *Debugging the bugs in buggy code*

What programming is mostly about is not writing new code so much as debugging[8] what you have already written. Key then is to reduce the incidence of bugs occurring in the first place, and when they do occur, firstly to have code that lends itself to debugging and secondly, knowing how to go about the debugging. The first two facets are at least partly addressed through good programming practice (see earlier)[9].

[8] The art of fault-finding in computer code.

[9] And by the discipline of software engineering, which is way out of scope of this course.

Here's an example to try out to start to see what might be involved in debugging, loosely based on a previous plotting example – go create a new **m-file** called: **plot_some_dull_stuff.m**[10]. Then add the following lines to the file:

[10] Remember – you are advised to name your `m-files` as something vaguely descriptive of what the script actually does (and you do ont have to go with this choice, although it might turn out to be perfectly descriptive ;) (i.e. you do not have to call it this!)

```
% my dull plotting program
% first, initialize variables and close existing figure
windows
close all;
x = -2*pi:0.1:2*pi;
y1 = sin(x);
y2 = cos[x];
% open a figure window and plot a sine graph
figure;
plot(x,y1,'r');
% add a cosine graph
hold on;
plot(x,y2,k);
```

and then run it (refer to above for how).

Pretty dull stuff eh? Wait – maybe you didn't get a figure appearing on the screen with a pair of sines and cosines on. Has **MATLAB** given you an error? If you typed in the above 'correctly', you should see:

```
Error:  File:  plot_some_dull_stuff.m Line:  6 Column:  9
Unbalanced or unexpected parenthesis or bracket.
```

Actually ... if this were your program, you should have paid attention to earlier and not have written it all at once before testing it! But at least **MATLAB** is giving you some sort of feedback. The actual error reported might not always mean that much to you but the line number at which the problem occurred is gold-dust. The line of code is does not like is line 6[11], which is:

```
y2 = cos[x];
```

[11] Note that although **MATLAB** ignores comment lines (in the context of executing code), it does count them when telling you which line of the program code an error occurs at.

Maybe the mistake is already obvious? If it is – go fix it and re-run the program. If not, maybe test out the line more simply, passing in a value directly to the function `cos` and not bother assigning the result to a different variable, e.g.

```
» cos[0.0]
```

to which you get told:

```
» cos[0.0]
 cos[0.0]
    ↑
Error:  Unbalanced or unexpected parenthesis or bracket.
```

Now you have reduced the use of the `cos` command to its simplest, whilst retaining the usage in your program that seemed to cause an issue. Hopefully, now the error is apparent. If still not, check out help on the `cos` function, or search `cos` in the **MATLAB** help (from the question mark icon in the toolbar).

Is it important to recognise that (1) bugs will not always be flagged by MATLAB with a line number, and you can have valid code but nonsensical results, and (2) the mistake is often made earlier in the code than when MATLAB flags up a problem line.

Other strategies for helping debug include:

1. Checking the what the values of the variables were at the point at which the program derped – the current (and the point of program crash) variable values are listed in the **Workspace window**.
2. Changing the relevant variable value(s) (here `x`) and re-typing the problem line to see if it makes a difference[12].
3. Commenting out (%) lines of code temporarily, or adding in additional (temporary) lines of code, and re-running. Where coding in bite-sized chunks is an advantage in this respect, is that if a program stops working after you have added a new section o code, you can go comment out the new code (never normally just delete it all), check that the original section of code still works, and then line-by-line, un-comment the new code until the problem line is found.
4. You can also put your program on hold just before the problem line and explore the state of the variables at that point (see Box), although in this particular example of a bug, MATLAB does not allow this, presumably because if feels that the mistake is simple and can be easily fixed.

[12] This is sort of similar to the example given of simply testing a specific value directly.

Once you have fixed this, re-run the program. Ha ha – it still does not work. (It is far from unusual to have multiple mistakes in the same piece of code, hence why writing the code in chunks and testing each time is helpful.) Now we have a problem on line 12:

```
Undefined function or variable 'k'.

Error in tmp2 (line 12)
plot(x,y2,k);?
```

Now **MATLAB** does not like `function or variable 'k'` because it cannot find that it has ever been defined. Is `k` meant to be a function or variable? Look up `help plot` to remind yourself of the correct syntax if the problem is not immediately obvious.

Once you have fixed the second bug; saved, and re-run the script, you should see Figure 3.1.

Debugging – breakpoints

Breakpoints are indicators in the code that tell MATLAB to pause that that point. This allows for in-depth testing of variable values and lines of code without having to exit the program.

To add a breakpoint in the code – click in the (grey) margin of the code editor on the problem line or before, and MATLAB adds a red circle to indicate a 'breakpoint' has been set. The presence of a breakpoint tells MATLAB to pause that that line.

To unset a breakpoint, click on the red circle or you can clear one or more from the drop-down **Breakpoints** menu in the toolbar.

3.2 *Functions*

Functions in **MATLAB**, are really just fancy scripts. Again – just plain old lines of code in a text file that is given a **.m** extension (making

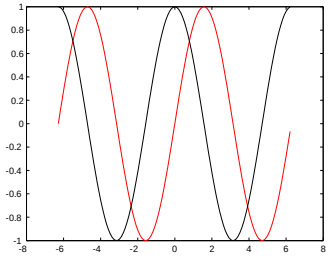

Figure 3.1: Output from the (bug-fixed version of) `plot_some_dull_stuff` **m-file**.

it an **m-file**). The big difference from a *script* in MATLAB is that a *function* can take variables as input and/or return an output (in contrast, a script takes no input and returns no outputs, other than plots or data files that might be saved).

A *function* is defined (and differentiated from a *script*) by a special line at the very start[13] of the m-file (see Box).

This is all not as weird as you might think. For example, you have already used the function `sin` – this takes a single input (angle in radians), and returns a single output (the sine of the angle). If you were to write your own function for `sin`, the file would start something like:

```
function [Y] = sin(X)
```

You can't, of course, go re-defining pre-defined MATLAB function names[14]. So how about if in your work, you found you frequently needed to use the square of the sine of a number. You could keep writing:

```
Y = (sin(X))^2
```

or, if you were a little more devious, you could create your own function for returning the square of the sine of a number. Your **m-file**, which here we'll call `sin2`, the contents of which would look like:

```
function [Y] = sin2(X)
Y = (sin(X))^2;
end
```

but of course with LOTS of comments to remind you what the function does etc. The new *function* is used pretty much as you would expect and have used previously, e.g.

```
» sin2(0.5)
```

will return the square of the sine of a value of 0.5 and dump the answer to the screen, and

```
» Y = sin2(0.5);
```

does the same but assigns the answer to the variable Y (and the semicolon suppresses output to the screen).

Go create your own function now. Start by creating one that takes a single input and returns a value equal to the sine of the square of the value (rather than the square of the sine as above). When you are happy with this, create one with 2 inputs (see Box), that returns a value equal to the sine of the first input, divided by the cosine of the second input[15] (i.e. $y = \frac{sin(x_1)}{cos(x_2)}$).

You have used other functions, perhaps without knowing it, and some of them return values, but because you have not attempted to

[13] Literally: line 1. Not even a comment line is allowed to appear before the *function* definition line.

Functions

The all-important fancy first line of a *function*, as defined in MATLAB help, looks like:

```
function [y1,...,yN] =
myfun(x1,...,xM)
```

Thanks MATLAB (this seems overly complex to say the least)!

OK – lets break this down. Lets assume that you call the **m-file** calc_stuff. The minimal definition of a function then looks like:

```
function [] = calc_stuff()
```

(The syntax is critical and the definition line must look like this.) Here we are saying – pass in not parameters and return no values either. So exactly like a normal script would work and you would execute the function calc_stuff by typing at the command line:

```
» calc_stuff()
```

(Maybe you can get away without the `()` bit.)

If you wan to pass in a single parameter (here: X), then you define the function:

```
function [] =
calc_stuff(X)
```

(To pass in more than 1 variable, simply comma separated the variable names.)

To pass out a parameter (here: Y) (and no input):

```
function [Y] =
calc_stuff()
```

Lastly, at the end of the function, you include the line:

```
end
```

[14] Actually you can, but it is best not to.

[15] Mathematically, the answer is not valid for all possible values of the 2 inputs (why?), and later we'll learn how to pro-actively deal with such a situation.

assume the returned values to anything, you have not noticed. `plot` is just such an example and if you look up » `help plot`, you'll see (towards the end of the help text):

```
plot returns a column vector of handles to lineseries objects,
one handle per plotted line.
```

Finally, it is important to note that by default, any variables created within a *function* are TOP SECRET, and by that, I mean that they are not accessible to the main **MATLAB** workspace and do not appear listed in the **Workspace window**. To see that this is a non-Trumpable true fact, create the following function (basically, the first example but split into 2 steps):

```
function [Y] = sin2new(X)
tmp = sin(X);
Y = tmp^2;
end
```

Here, a variable `tmp` is created to hold the value of the partial calculation. It does not appear in the **Workspace window** when you use the function. The advantage of this is that you could create a second function that also created a temporary variable internally called `tmp` with both instances of `tmp` treated entirely sperate and isolated by **MATLAB** (i.e. setting the value of one instance of `tmp` does not affect the value of the other). This also however does lead to some additional complications in debugging *functions* (see Box). Try setting a breakpoint at the start of the line where the square of `tmp` is calculated – note that `tmp` now appear in the **Workspace window**. Continue the function and when it terminates, note that `tmp` is now gone from the list.

Debugging – *functions*
Functions are a prime example of the importance of being able to pause code part the way through (e.g. by setting a breakpoint) because when a function terminates, or crashes, you get to see none of the values of any variables created within the function, unless they have been returned as output (and assuming here that the code did not crash and managed to get to the end). Setting a breakpoint allows you to interrogate the values of any internal variables.