2
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 ...

2.1 *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 (the list of commands) 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 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 are going to use the function `disp` (see margin help box and/or type » `help disp` to find out the **MATLAB** *function* syntax and usage). This command (/*function*) will print to the screen, either any text you specify (in inverted commas), or the contents of a *string variable* (you pass the variable name to `disp`). 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

[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 file 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.

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

`disp`
... displays something (the contents of a variable) to the screen.
In the example of:

```
disp('STRING')
```

where `STRING` is a .... *string*, get the *string* displayed as text at the command line.
You can also pass the name of a variable that contains a string, e.g.

```
disp(VARIABLE)
```

where the contents of `VARIABLE` is a string.
Note that the difference between using `disp` and simply typing the variable name:

```
disp(X)
```

is ... well, find out for yourself!
Note that in some situations, its effect is simply the same as leaving off the semi-colon (;) from the end of a line.

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)
```

(Try this out.)

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).

[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.).

2.1.1 *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).
- 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* line).
- 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;
```

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.

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

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

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] = 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
```

The schematic for the program structure is shown in 2.1. (Don't worry what this particular program does, just note how I have structured it.)

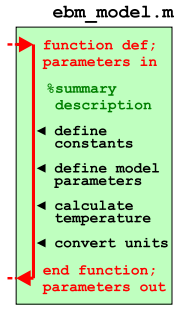

Figure 2.1: Schematic of the example program.

This example 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).
Also note, if it helps – you can run a single line of code over 2 lines of the file by adding

```
...
```

at the end of a partial line (that is to be treated by **MATLAB** as joined continuously to the next 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 ***
```

The schematic structure of this program (*script*) is shown in Figure 2.2.[7]

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[8].

[7] Note that not all of the comment lines are shown in the structure schematic – only the main program summary at the top.

Figure 2.2: Schematic of the Hello World program.

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

2.1.2 *Debugging the bugs in buggy code*

What programming is mostly about is not writing new code so much as debugging[9] 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)[10].

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[11]. Then add the following lines to the file:

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

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

[11] 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 earlier 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[12], which is:

[12] 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.

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

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 derp-ed – 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[13].
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.

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

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.

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 apparently 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 2.3. (unless there were further bugs to find ...)

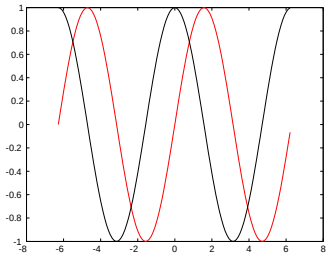

Figure 2.3: Output from the (bug-fixed version of) plot_some_dull_stuff *m-file*.

2.2 *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 it an *m-file*). The big difference from a *script* in MATLAB is that a *function* can take variables as input and/or return variables (or variable values) as 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[14] 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[15]. 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.

In this example, the contents of your *m-file*, which here we'll call `sin2`[16], 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.

Your 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 command line, and

```
» Y = sin2(0.5);
```

does the same but assigns the answer to the variable Y (with the semi-colon suppressing output to the command line).

Go make up your own *function* now. Start by creating one that takes a single input and returns a value equal to the sine of the

[14] 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
```

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

[16] And hence filename sin2.m.

square of the value (rather than the square of the sine as above). Test it (i.e. compare the output of your *function* with the equivalent calculation typed in at the command line).

When you are happy with this, create one with 2 inputs (refer to **MATLAB** `help` on `function` and/or refer to the previous Box), that returns a value equal to the sine of the first input, divided by the cosine of the second input[17] (i.e. $y = \frac{sin(x_1)}{cos(x_2)}$).

[17] 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.

You have used other functions, perhaps without knowing it, and some of them return values, but because you have not attempted to assign the returned values to a variable, you may not have not noticed. For example, `plot` and `scatter` are in fact *functions*, and return an ID of the plot graphic. We simply have not been asking for the returned value so far. As per **MATLAB** `help`:

```
H = SCATTER(...)  returns handles to the scatter objects
created.
```

with the `handle`, `H`, being an identifier of the graphic which could prove to be useful if e.g. you would like to modify one of the properties of an existing graphic.

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-Trump-able 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, we have created a variable `tmp` 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).

The private nature of *variables* created within *functions* does however does lead to some additional complications in debugging *functions* because when the function terminates, you have no record of what occurred during its execution (in terms of not being able to access the value of any of the variables used within the *function*). Try setting a breakpoint at the start of the line where the square of `tmp` is calculated – note that `tmp` now appears 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*.

2.3 *Conditionals '101'*

2.3.1 `if ...`

One of the most important programming constructs is the *conditional statement*, in which whether one or more *statement*(s) are executed (and hence the overall outcome) is conditional on the 'truth' or otherwise (i.e. it being true or false) of a given *expression*.[18]

This is embodied in **MATLAB** (and similarly in most languages) by the `if ... end` construct (see *Conditional Statements* Box).

In creating an `if ... end` construct, the statement tested for truth can be any one of:

1. A *variable* having a value of `true` (`1`) or `false` (`0`). e.g.

   ```
   if happy
   ...
   ```

   where `happy` is a variable.
2. A **MATLAB** *function* returning a `true` or `false`, e.g.

   ```
   if isnan(A)
   ...
   ```

   where variable `A`, may or may not be a `NaN`.
3. A *relational operator* (see earlier), i.e. one of e.g.:

   ```
   >, <, <=, >=, ==, ~=, &&, ||
   ```

   and applied to a pair of *variables*, one *variable* and one value, or two values, e.g.:

   ```
   if A > B
   ...
   ```

   where `A` and `B` are numbers.

All this will hopefully become apparent during this and later weeks, so don't worry about the details ... just yet.

An initial and rather computer programming textbook-like example is as follows:

Designing a program (a **MATLAB** script saved as an *m-file*) that asks whether or not you like bananas, and if you answer 'yes', tells you 'Correct – they are a great fruit!'.

But before we worry about anything else (e.g. how to apply a *conditional* statement), you'll need to know about inputting information into a **MATLAB** program from the keyboard[19]. Amazingly, you can guess (I actually just did) the command for requesting input – it is `input` (for 'input' – a rare occasion when everything is logical and simple!) (see Box).

[18] Pause ... and deep breath.

Conditional Statements

The principal *conditional statement* in **MATLAB** is: `if ... end`

The basic `if` structure is:

```
if EXPRESSION (IS TRUE)
   STATEMENT(S)
end
```

in which the code `CODE` is executed if `EXPRESSION` is evaluated as `true`. No code is executed otherwise (and `STATEMENT` is `false`).

A variant addition – `else` – which allows for an alternative block of code (`OTHER STATEMENT(S)`) to be executed if `EXPRESSION` is instead evaluated as `false`, is:

```
if EXPRESSION (IS TRUE)
   STATEMENT(S)
else
   OTHER STATEMENT(S)
end
```

Finally, there is 3rd variant including `elseif`:

```
if EXPRESSION (IS TRUE)
   STATEMENT(S)
elseif EXPRESSION (IS
TRUE)
   OTHER STATEMENT(S)
else
   OTHER STATEMENT(S)
end
```

Now, assuming that the first **EXPRESSION** is not true, a second **EXPRESSION** is evaluated, and only if that second **EXPRESSION** is also not true, will the final possible **STATEMENT** be evaluated. (Here, this final variant is shown with an `else ...` included at the end, but this is not a formal requirement to include.)

[19] All programming languages have such a facility and man basic programs, at least in the Old Days prior to widespread GUIs, make use of keyboard input

Armed with this important new information (how to get **MATLAB** to ask for input and then receive and do something with keyboard input) – firstly create a blank *m-file* and save with a 'suitable' filename. Maybe add a header comment (1st line or lines starting with a `%`) to remind you what this *script* is going to do.

Secondly, (and on the next line) – define the text (question) that you are going to ask and assign this string to the variable `MY_QUESTION` (substitute your own filename here). Then place the input `command` (on the next, now 3rd line) for string input, and assign the input string to the variable `MY_ANSWER`. You should have a program consisting of 3 (or more, depending on how much commenting you do) lines – an initial comment line, a line defining the question and assigning this string to a handy variable (`MY_QUESTION`), and a line taking the results of the input function, and assigning it to a second variable (`MY_ANSWER`). The structure of your program should look like Figure 2.4. To help you out, a complete program looks like:

```
% === a program to ask whether I like bananas ===
% first - specify the question (and assign to a variable)
var_question= 'Do you like bananas?';
% now ...  ask the question!
var_answer = input(var_question,'s');
```

Run the program thus far. You should see the question displayed, and when you type in an answer and hit **RETURN**, the program will end. Because your *m-file* is configured as a *script* and not a *function* (see earlier), you can see the variable `MY_ANSWER` in the variable list and you can hence check its value – it should contain a *string* with the answer you gave to the question. Make sure it all works like this so far.[20]

OK – aside from the use of `input`, there is nothing new here. Yet. The ultimate purpose of the program is to give a reply that depends on the answer given. This is where we are going – to utilize a *conditional statement* – depending on whether the answer is 'yes' or not, we are going to display a different message. This is a fundamental programming element – different code (the *statements* in the *conditional* definition) will execute depending on the value of a *variable* – in this example, the 'different code' is a different message and the value of the variable is 'yes' or 'no' (or other answer).

You are going to add an '`if ...`' statement to the code (starting on line 4) to test whether the answer, held in the variable `MY_ANSWER`, is equal to '`yes`'. In the language of **MATLAB** syntax (see Box), the EXPRESSION is whether the string contained in `MY_ANSWER` is '`yes`'. How do we ask **MATLAB** to compare the value of `MY_ANSWER` with

`input`

There are two variants – one for inputting numerical information and one for inputting a string (as `1` could be either the value one or a 1-character string ...).

For inputting a numerical value:

```
X = input(PROMPT)
```

will display the text in the string variable `PROMPT` and set the value of variable `X` to whatever number is entered (and after RETURN is pressed).

For inputting a string:

```
STR = input(PROMPT,'s')
```

will display the text in the string variable `PROMPT` and set the value of `STR` when a string is entered (and after RETURN is pressed). Note that the second parameter passed to the function `input` (`'s'`), tells **MATLAB** that the input is a string rather than a number.

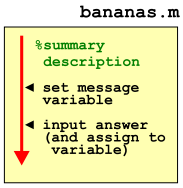

Figure 2.4: Schematic structure of the simple bananas question program.

[20] HINT: When you type the answer, it appears on the screen immediately adjacent (and untidily) to the end of the question. You can make this look nice(r) by adding a space at the end of the question string you assigned to prompt, e.g. `PROMPT = 'Do you like bananas? ';`.

'yes'?

Once upon a time, long long ago, **MATLAB** was simple and helpful and you could write:

```
if (my_answer == 'yes')
  [MESSAGE]
end
```

where `[MESSAGE]` you will later replace by a message that you will display using the `disp` command that you saw before. (In this stupid example it might be: 'Correct – they are a great fruit!').

However ... life is no longer this simple. **MATLAB** is going to make us use the function `strcmp` (see Box). In using `strcmp` we might break things down into 2 steps – the first comparing the 2 strings (`MY_ANSWER` and `'yes'`) and returning to us a value of *true* or *false* that we will store in a new variable. In the second step, we'll ask the conditional to act on the value of the variable. The code will now look like this:

```
COMPARISON_RESULT = strcmp(MY_ANSWER, 'yes');
if COMPARISON_RESULT
  [MESSAGE]
end
```

Or, we could have made this more compact:

```
if strcmp(MY_ANSWER, 'yes')
  [MESSAGE]
end
```

Your code should now comprise something like the 3 lines from before (comment, define question, get input) followed by 4 lines of code of the conditional structure, comprising: the `strcmp` function, the `if ...`, use of `disp` to display a message, and lastly, `end`. The structure should look like Figure 2.5[21] or if you assign the message to a 2nd variable, like Figure 2.6. A complete example program ... to help you follow all the above, would look like[22]:

```
% === a program to ask whether I like bananas ===
% === (and now give an answer!)  =================
% first - specify the question (and assign to a variable)
var_question = 'Do you like bananas?';
% second - specify the response (and assign to a variable)
var_response = 'Me too!  OMG I could die!';
% now ...  ask the question!
var_answer = input(var_question,'s');
% test the answer ...  and reply if 'yes'
if strcmp(var_answer, 'yes')
   disp(var_response);
end
```

`strcmp`

For once, the **MATLAB** help explanation is relatively simple and straightforward:

```
tf = strcmp(s1,s2)
compares s1 and s2 and
returns 1 (true) if
the two are identical.
Otherwise, strcmp returns
0 (false).
```

Which is pretty well much how we expected asking: `s1 == s2` to pan out.

(In **MATLAB help** – `tf`, the variable name used in the example, is short for 'true-false'.)

[21] The red triangle denotes a branch point, where the code can go in different directions depending on the result of the *conditional*. In this example – there is only one branch, corresponding to the answer being 'yes'.

[22] Note the indentation of the contents of the `if ... end` structure. This is very common programming practice. You can make **MATLAB** do this for you by selecting a single line, or highlighting a block of lines, and clicking on the `Indent` icon in the code editor.

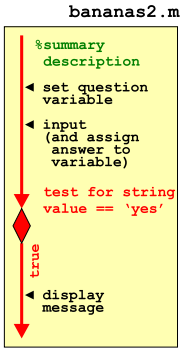

Figure 2.5: Schematic structure of the extended bananas question program.

(Please – do not just copy-paste the code ... write your own code and only use, if you really need it, this code as a guide.)

Re-run (after saving) the program and confirm that it works (asking whether you like bananas and if you answer 'yes', tells you 'Correct – they are a great fruit!'). If not – time to de-bug! Note that if you tested the code in two stages, any bug at this point is only in the conditional structure. Start by double-checking the syntax required for the `if ...` structure. You could also try commenting out the message line and re-running.

Next, you might display an alternative message is the answer is not 'yes'. Refer to **help** / the margin Box on if ... and note that you can extent the structure with an `else` which would be followed by a line displaying the alternative message (e.g. 'Then you need to get a life, apple-lover.')[23].

Try this first – extend you program with an `else` line and then a an alternative message. The structure should now look like Figure 2.7.

You could also turn this around, and test for any answer except 'no' (the ~ is making the test, not 'no'), i.e.

```
if ~strcmp(MY_ANSWER, 'no')
  [MESSAGE 1]
else
  [MESSAGE 2]
end
```

Now you are asking whether the answer is something other than 'no' (which might be 'yes', but not necessarily so) – in the logical construct – whether the (string) contents of `answer` are not equivalent to 'no'.

Finally – you could extend this example further and tackle the situation of their being 3 possible answers – 'yes', 'no', and ... 'I don't know' (or any other answer). Now the basic structure becomes

```
if strcmp(MY_ANSWER, 'yes')
  [MESSAGE 1]
elseif strcmp(MY_ANSWER, 'no')
  [MESSAGE 3]
else
  [MESSAGE 2]
end
```

Here – we are now adding an `elseif ...` line (followed by its specific message) (and see Box/**help**). Maybe try this and test it fully – inputting a 'yes', a 'no', and some other answer, and confirming that you get the correct message displayed.

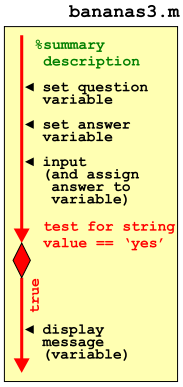

Figure 2.6: A slight variant on the schematic structure of the extended bananas question program.

[23] And then the line with `end` after that – follow the prescribed structure *exactly*.

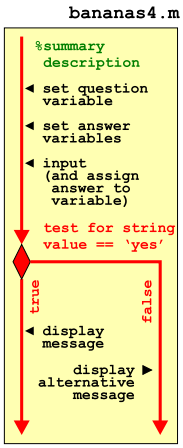

Figure 2.7: Schematic of the bananas program using the `if ... else ...` construct (and displaying alternative messages).

Continuing to beat this same tired example to death ... what if some wise-crack answered 'YES' rather than 'yes'?[24] One could write:

```
if strcmp(MY_ANSWER, 'yes')
  [MESSAGE 1]
elseif strcmp(MY_ANSWER, 'YES')
  [MESSAGE 1]
end
```

This will work, but you might note that you have had to exactly duplicate the MESSAGE line. If instead of displaying a simple message, a complex calculation was carried out – all the lines of the code following the `if ...` would have to be exactly duplicated after the `elseif ...` . While it might seem trivial to simply copy-paste the required lines, this is[25] dangerous – if the first set of lines are ever changed (due to a bug-fix or simple further development of the code), the same changes MUST then be exactly duplicated in each and every instance, or the code will not longer work correctly. This is *very* easy to forget to do, particularly for extensive code or code that you have not looked at for ... years. Code duplication also makes the overall code unnecessarily long (and hence harder to look through).

Instead, we can nest statements containing relational operators. What does this mean? Well, in the example of the answer being 'yes' or 'YES', logically, what we want is:

(1) the contents of `answer` is equivalent to 'yes'
    OR
(2) the contents of `answer` is equivalent to 'YES'

In code, this is written:

```
strcmp(answer, 'yes') || strcmp(answer, 'YES')
```

Make sure you are happy with what this means (it is pretty well much exactly as it looks == logic).

So – go modify your code to allow for a 'YES' or a 'yes'. Hell, try allowing for a 'Y' or a 'y' as well.[26] (You could extend it to 'no' also but I think you get the point ...)

A non-text and non fruit related example. Almost.

How many bananas could you eat in a day? I bet it is less than ten. We'll let the computer ask and if the answer is 10 or more, you (the computer) shouts: 'lier!'.[27]

[24] This goes to the heart of all software testing – what if the user does something you were not expecting? Hence why all software undergoes extensive testing by user or people who did not test it. Sometimes there are pre-releases ('alpha' or 'beta' versions or simple 'pre-release') of software to all or specific parts of the user community, precisely to provide feedback, find bugs, and see whether they can break it ...

[25] Note quite in the same way that driving down a mountain highway with your eyes shut or hungry sharks are dangerous.

[26] Sort of for this reason and that there are many different ways of writing 'yes', software often requires you to answer 'yes' in a restricted number of ways – this restriction is made clear as part of the message that asks the question. Common is to restrict the answer to 'Y' or 'y'.

[27] This example is even more stupid than the last one. But no more stupid than in any computer programming textbook and it will at least demonstrate a subtly different usage of `if ...`.

The basic code is very similar to before. Create a new *m-file*, add a comment line, define your question ('How many bananas do you think you could you eat in a single day?') and then get **MATLAB** to ask it and pass back whatever is entered in at the command line. The only difference at this point – refer to the usage of `input` (see earlier Box) – is that we want a number input rather than a string. You can call the *variable* into which you assign the result of `input`, the same as before, or to make it distinct, e.g. `N_BANANAS`, i.e.

```
N_BANANAS = input(MY_QUESTION)
```

In the if statement, we now want to test whether the value of `N_BANANAS` is greater or equal to 10 (or equivalently, greater than 9), i.e.

```
if (N_BANANAS >= 10)
   [MESSAGE 1]
else
   [MESSAGE 2]
end
```

or equivalently:

```
if (N_BANANAS > 9)
   [MESSAGE 1]
else
   [MESSAGE 2]
end
```

Write this code and get it going. Feel free to switch fruit / fruit consumption threshold, question/answers, or whatever.

2.3.2 *switch ...*

A less commonly used alternative to `if ...` is `switch ... case ...` and is helpful in the case of multiple possible correct answers and/or multiple different answers.

For instance, and back to the ... fruit ... you might want the same answer for multiple different kinds of fruit. Trying coding up the program that would give you 'A great fruit!' for any of 'banana', 'kiwi', 'apple', 'pineapple', and 'cucumber' (yes they are technically fruit – Google it). You will find either you have many lines of code and many duplicated lines of the same message, or a very long line after `if ...` with loads of `strcmp` and ORs (`||`). Using `switch ... case ...` the code instead might look like:

```
switch MY_ANSWER
   case {'banana', 'kiwi', 'apple', 'pineapple', and 'cucumber'}
      disp('A great fruit!')
   otherwise
```

Conditional Statements (2)

The other main *conditional* statement is: `switch ... case ... end`

The basic `switch` structure is:

```
switch VARIABLE
   case VALUE(s)
      STATEMENT(s)
end
```

which deviates rather from how **MATLAB** describes it, but this makes more sense to me (and hopefully to you). Here, `VARIABLE` is a variable and it is compared with one or more `VALUE(s)`. If the value of `VARIABLE` matches that of the `VALUE(s)`, then `STATEMENT(s)` are executed.

A common variant adds a default set of `STATEMENT(s)` to be executed if the value of `VARIABLE` does not match any of the `VALUE(s)`, e.g.

```
switch VARIABLE
   case VALUE(s)
      STATEMENT(s)
   otherwise
      STATEMENT(s)
end
```

You can also have multiple case possibilities:

```
switch VARIABLE
   case VALUE(s)
      STATEMENT(s)
   case VALUE(s)
      STATEMENT(s)
   otherwise
      STATEMENT(s)
end
```

```
        disp('yuck!')
end
```

where `MY_ANSWER` is the name of a fruit entered in, in response to input, e.g.

```
MY_ANSWER = input('What is your favourite fruit?,'s');
```

Note that for a list of multiple possible value, **MATLAB** requires the list after `case` to be encased in {}. For a single answer, it would just be:

```
    case 'banana'
```

for a string, and for a number:

```
    case 10
```

2.4 *Loops '101'*

The next main program construct that you are going to see is the *loop*. There are a number of different forms of this in **MATLAB** (see *loops* Box) (and also in other programming languages), but the basic premise is the same – a designated block of code (one of more lines of code[28]), is repeated, until some condition is met. That condition might be something as simple as a count having been reached, e.g. the block of code is always executed n times, or the condition might be slightly more complex and involve a *conditional statement* (see later). Will explore a very basic loop though an example, almost as contrived as for conditionals :o)

2.4.1 `for ...`

In this subsection we'll start with a very straight-forward and somewhat abstracted usage of `for ...`, which hopefully will get you in the mood for *loops*. Then we'll go through some slightly more problem-focused examples.

Loops Ground Zero. Basically – `for` *loops* cycle through a series of numbers between specific limits, or if you like, 'count' up (or down) through a series of numbers. As the loop counts (cycles), it allows you to execute some code, so for each count (or cycle), the (same) block of code is executed. We'll worry about what you might 'do'[29] (i.e. the code fragment) in a *loop*, later.

Consider, or rather: create a new m-file[30], and add the code for the following loop:

```
for n=1:10
end
```

Save it. Run it. What did it do?

I bet you have absolutely no idea! It actually cycled around ten times, counting from `n=1` through `n=10`, but you would not know it as there was no code within the loop to do anything and tell you anything about it.[31]

There are 2 alternative but very crude debugging strategies you could take[32]:

1. Simply add a line within the loop with the name of the (counting) variable, e.g.

```
for n=1:10
   n
end
```

***loops* in MATLAB**

`for`

The basic `for ... end` structure is:

```
for n = VAL1:VAL2
   CODE
end
```

where `VAL1` and `VAL2` are the limits that `n` will count between (starting at `VAL1` and ending at `VAL2`), meaning that `STATEMENT(S)` will be executed `(VAL2-VAL1)+1` times in total. `STATEMENT(S)` can be one or more lines of code, that will all be executed on each and every cycle of the loop.

The loop need not count in increments of one (1), the default, e.g.:

```
for n = VAL1:INC:VAL2
   CODE
end
```

counts with an increment of `INC`. It is also possible to count down (a negative value of `INC`).

`while`

The basic structure is similar to that for `for ... end`:

```
while STATEMENT (IS TRUE)
   CODE
end
```

`while` differs from `if` in that there are no alternative branches of code that can be executed. The `while ... end` loop cycles and `CODE` continued to be executed (for ever) until the `STATEMENT` is evaluated to be `false`.

[28] It is possible to for the block of code to be only a fragment of a single line and hence the entire *loop* plus code block, to be written on a single line.

[29] Note intentionally a joke. Actually, this is only funny if you know **FORTRAN**, and even then it is only marginally funny.

[30] Comment it!

[31] You get one clue – if you look in the variables Workspace window, you'll see there is a *variable* `n`, with a value of `10` – the last value it was assigned before the *loop* ended.

[32] Plus, you could add a *breakpoint* and view the value of `n` in the Workspace window each cycle around the loop.

and it will spit out the value of `n` each time around the loop.

2. Print the value of `n` 'properly'[33], e.g.

```
for n=1:10
   disp(n)
end
```

or

```
for n=1:10
   disp(num2str(n))
end
```

or you can tart this up even nicer by creating a string that provides more explicit information back to you, which is when you really need to use `num2str`, e.g.

```
for n=1:10
   my_string = ['The value of n is:  '  num2str(n)]
   disp(my_string)
end
```

or if you are happy with more going on in a single line:

```
for n=1:10
   disp(['The value of n is:  '  num2str(n)])
end
```

(but they work the same – check it).

If you are not yet 100% with *concatenation* – the 'action of linking things together in a series' (dictionary definition), what is happening in the line:

```
my_string = ['The value of n is:  '  num2str(n)]
```

is that you are taking the string `'The value of n is:  '`, and the string equivalent of the numerical value of `n` (created via the use of `num2str`) and ... joining them together, one (`num2str(n)`) after the other (`'The value of n is:  '`).

Loops in action. So, consider the following (contrived) problem – you want to be able to enter a series of numbers and return their sum (although equally one could perform and return all sorts of statistics).[34] The basic code is simple and you can try it out by first creating a new (*script*) m-file.

Using the other (numerical input) form of `input` (see earlier), the code fo entering 2 numbers, one after the other, might look like this (although in practice, your code is full of helpful comments, right?):

```
my_question = 'Please enter a number:  ';
A = input(my_question);
my_question = 'Please enter a number:  ';
```

[33] Although you can get away with just writing:

```
disp(n)
```

[34] Obviously, one way to do this would be to enter the numbers into a file first, use the load function, and calculate the sum.

```
B = input(my_question);
disp(['The sum of the numbers is:  '  num2str(A+B)]);
```

The first 4 lines you should be A-OK with, you have seen something very like this before. I the last line, again, 2 strings have been concatenated by enclosing `'The sum of the numbers is:  '` and `num2str(A+B)` in a pair of brackets `[]`. The string representing the number sum is itself created by adding `A` and `B`, and then converting the resulting number into a string using `num2str` (see earlier). As always – if you are happier breaking down the last line into its component parts, e.g.

```
answer = A+B;
answer_string = num2str(answer);
disp(answer_string);
```

then please do! There is no particular computational penalty in MATLAB (at least, not at this stage) for creating as many variables as you like and breaking down code into multiple lines.

So far so good. But what if you wanted 4 numbers summed ...

```
my_question = 'Please enter a number:  ';
A = input(my_question);
my_question = 'Please enter a number:  ';
B = input(my_question);
my_question = 'Please enter a number:  ';
C = input(my_question);
my_question = 'Please enter a number:  ';
D = input(my_question);
disp(['The sum of the numbers is:  '  num2str(A+B+C+D)]);
```

You can see whether this is going – firstly that you are duplicating more and more lines of code as the number of numbers increases. Secondly, and we'll come to that in a moment – what if the program does not know *a priori* how many numbers you want to sum? Or do you need to write a program for every single possible number of numbers that you might need to input and process? An impossible and thankless task ...

You can see the code that is being repeated (here for input `x`):

```
my_question = 'Please enter a number:  ';
x = input(my_question);
```

If you bothered to read the margin box earlier, you'd known that this is exactly what a *loop* can be used for. We therefore want something of the form:

```
for n = 1:MAX_N
   my_question = 'Please enter a number:  ';
   x = input(my_question);
end
```

The easy part is the configuration of the loop – in the previous example with 4 inputs, we would write:

```
for n = 1:4
```

and the *loop* with go around 4 times as the counter `n` counts from 1 to 4 (`MAX_N`) in increments of 1 (the default behavior of the *colon operator*). Each time around the *loop* the block of (2 lines of) code is executed and a number is inputted. But what is still missing? Try it exactly like this and see if you can see what is going on, or rather, not going on. If you think it is not working as expected – try some debugging (i.e. adding one (or more) `disp` statements within the loop code, or add a *breakpoint* within the *loop*). See if you can come up with a solution once you see what the problem is. (Warning: the spoiler is in the margin.)

After having tried your own solutions, try out both of the given alternatives (see margin) (assuming that one of them was not also your solution). Note that you are not given the complete code needed and some further debugging might be needed (but they do both work!).

Two things to be aware of in doing this:

1. If you set the maximum number of items quite high and then get bored and need to exit the program – press the key combination Ctrl-C and **MATLAB** will exit your program (but leave **MATLAB** running).
2. If you run the program a second time and use the vector approach, something very odd starts to happen to the reported sum. This is because there already exists a *vector* with the same name left over from the first time you ran the *script* program. You can solve this (first try it out – running the program several times in a row to see what happens) either by initializing the vector `y`, just like you did for `x` in the 1st solution, i.e.

   ```
   y = [];
   ```

   (before the loop starts, of course), or you can clear the workspace using `» clear all` (clears *all* variables), or clear just the problem variable (`y`) that will end up growing and growing and growing ... (`» clear y`).

A different and simpler way of looking at creating a running sum, or in the case below, incrementing the value of a variable within the loop is to consider creating an explicit counting variable, sperate form the loop counter. Recall:

It should be apparent if you tried it out, that the value of `x` at the very end of the program, is equal to the last value you entered. In other words, each time you go around the loop you are over-writing the previous entered value and end up with nothing to sum at the end. There are two (or more) possibilities to solve this:

1. You could keep a *running sum*. This would also avoid having to explicitly calculate a sum at the end, but you would not have saved the numbers as you went an no other stats would be possible.
   You would do this by adding the inputted value to the existing value, i.e.

   ```
   x = x + input(prompt);
   ```

   where `x` is the running total. What this says is: take the current value of `x`, add the value if the user input, and place the total back into the variable `x`.
   The only problem here ... is that MATLAB does not know what the very first value of x is – i.e. the value before the loop start and that you then try and add `input(prompt)` to. The solution is to initialise the value of x before the loop starts, e.g.

   ```
   x = 0;
   ```

2. Alternatively, you could add the newly inputted number to the end of an existing vector. In this way, you end up recording all the values that were inputted. e.g.

   ```
   y = [y input(prompt)];
   ```

   which says take the vector `y`, and add a further value (`input(prompt)`) to the end of it. At the end of the program (after the loop has terminated), you have to sum the contents of the vector `y`.
   Or, to break it down:

   ```
   z = input(prompt);
   y = [y z];
   ```

```
for n = 1:10
end
```

will simply loop around 10 times, as the *loop* counter `n` is repeatedly incremented by 1 (the default increment of the colon operator), until it reaches a value of 10.

Create a new m-file and enter the following code:

```
m = 0;
for n = 1:10
   m = m+1;
end
```

What do you expect to happen to the value of m? Add some `disp` statements and print out the values of `n` and `m` (from within the *loop*), each time around the *loop*. Was this what you expected? Why? What about the following?

```
m = 1;
for n = 1:10
   m = m+1;
end
```

Or ... what do you expect in this:

```
m = 2;
for n = 1:10
   m = m^2;
end
```

As abstracted and odd as it might seem now, later, this will all be important to understand. Please make sure you do!

You might note that you should not substitute the variable name `n`, for `m`, i.e. as in something like:

```
n = 0;
for n = 1:10
   n = n + 10;
end
```

Why? (Try it and see, even.)

2.4.2 *Other loop configurations and usages*

In the previous examples, the *loop* limits were fixed in the program itself – you'd have to edit the *script* code and re-save the file in order to be able to input and sum a different number of values. You could create a more flexible program by making the m-file a *function* rather than a *script*.[35]

[35] There are other ways of adding flexibility to the loop count that we'll see shortly.

The idea here is to create a *function* that takes a single input. This input will be the maximum *loop* count. If the input variable was called `max_count`, then the *loop* structure would now look like:

```
for n = 1:max_count
   my_question = 'Please enter a number:  ';
   x = input(my_question);
end
```

Referring to the previous lessons on *functions* (as well as help if need be), create a *function* that when you call it, e.g. like:

```
» function_sum(5)
```

will request 5 inputs in turn, and at the end, display the sum.[36]

Then create a variant of this *function*, and have it return the sum, rather than display it. i.e. you *function* will now take as input, the number of numbers you wish to input, and will return the sum of those numbers.

Alternatively, your program (as a *script*), before the loop starts, could ask for the number of values to be entered, passing this to the variable `max_count`, with the loop then looking exactly like the above. In both cases you are substituting a fixed number (e.g. 4) for a variable that might contain any number.

[36] So in addition to the code fragment given, you need to define (at the top) and then end (at the bottom) a *function*, you need to create a running sum, and then after the *loop* finishes, display the sum.

Finally, in addition to a flexible *loop* count maximum limit, the value of the increment in the count each time around the loop need not be one and it also need not start from 1. For example:

```
for n = 10:10:100
   ...
end
```

is exactly equivalent in terms of the number of iterations carried out to

```
for n = 1:1:10
   ...
end
```

and which is the same as the default behavior of the colon operator:

```
for n = 1:10
   ...
end
```

The value of the loop counter `n` simply differs by a factor of 10 at every iteration between the top and bottom two versions.

2.4.3 *Fun(!) worked examples*

(Only one example to date. And not necessarily even fun.)

Loops, CAMERA, ACTION! (A more colorful example of *loops* in action.) What we are going to do is (load and) plot a sequence of monthly data-sets and put them together to create a movie (animated graphic) to illustrate the seasonality of temperature in global climate. You will hopefully thereby better appreciate the value of constructs such as *loops* in computer programming in saving you a whole bunch of effort and needless duplication of code. (Equally, you might not have

wanted a movie as the end result, but simply a number of plots, all identical except in the specific array of data they were plotted from.)

First download all the monthly global surface temperature data-files on the course webpage (there are 12 files to download)[37]. Then you are going to want to plot them all ... which would get desperately tedious if you had to do this at the command line 12 times. Think how much more of your life you would be wasting if the data were weekly. Or monthly data for 1972 through 2003, some 372 separate data-files ... You would never have time to go get a coffee ever again(?)

[37] In scripting, it is also possible to automate downloading files from the internet.

Create a new m-file. Call it ... anything you like[38]. However, as well as appropriately naming your script file, add a *comment* on the first line of the file as a reminder to yourself of what it is going to do.

[38] `bob_the_builder.m` counts as 'anything you like', but that looks pretty lame and it certainly won't help you remember what the script does if you came back to it sometime in the future.

To make an animation, we need to make a series of frames, with each one being a different monthly temperature plot (in sequence; Jan through Dec). The files are rather conveniently named: `temp1.tsv`, `temp2.tsv`, ... `temp12.tsv`[39]. We should start by loading this little lot in. For the first file we could write:

[39] Don't worry about the `.tsv` file extension – the file format is plain old text (ASCII) and could have instead been `.txt`.

```
temp = load('temp1.tsv');
```

or equally:

```
temp(:,:)  = load('temp1.tsv');
```

and hence with a slight-of-hand, we could also write:

```
temp(:,:,1) = load('temp1.tsv');
```

Can you see that these statements are identical? Run the script with one, then with the other, just to be sure. The last form is really useful, because we can now go on and write:

```
temp(:,:,2) = load('temp2.tsv');
```

What you have done here is to load the January 2D (lon-lat) temperature distribution into the 1st 2D layer of the `temp` array, and then we have gone and created a second 2D layer on top of the first with the February climate data in it. Look at the **Workspace window** (or type `size(temp)`) – you now have a 3D ($94 \times 192 \times 2$) array. Fancy! This is your first 3D array – there is nothing really conceptually different from the 2D arrays that you have already been using, we simply have a 3rd index for the third dimension – if it helps, you can think of a 3D array as being indexed by: row, column, layer.

You could go on and load in the March, April, etc data in a similar fashion, but you should be able to see a pattern forming here – each filename differs only in the number at the end of its name and this number corresponds not only to the number of the month, but will

also correspond to the layer index of the 3D array that you will create. This is something that a *loop* could be used for while you go off for a coffee. So this is what we are going to do – use a *loop* to load in all of the files. So go back and delete the lines that load in the files, one-by-one.

We first need to construct the *loop* framework. We'll call the month number counter variable, `month`. Create a `for` *loop* (with nothing in it yet) with `month` going from `1` to `12`.[40] Refer to the course text (this document!), and/or the **MATLAB** documentation, and/or the entirety of the internet, if necessary. The syntax (and examples) is described in full under » `help for`. Save the script (m-file) and run it[41]. What happens? Can you tell?

[40] Don't forget to suitably comment what it is that the *loop* does with a line (or even 2, but don't write a whole essay) beginning with a %.

[41] Typing: the `m-file` filename without the extension.

One way of following what is going on as **MATLAB** executes the commands within a script is to explicitly request that it tells you how it is getting on. You can use the function `disp` to help you follow what the program is doing (this is Old School debugging[42]). Within the loop, add the following line:

```
disp(month)
```

[42] You can also add a *breakpoint* within the *loop* and thus can cycle through the *loops* one-by-one, thereby being able to check the status of the variables within the loop and how they change from iteration to iteration.

then save and re-run the *script*. Now you can see how the loop progresses. This sort of thing can be useful in helping to *debug* a program – it allows you to follow a program's progress, and if the program (or **MATLAB** script) crashes, then at least you will know at what loop count this happened at, even if you are not given any more useful information by **MATLAB**. Only when you are happy that you have constructed a *loop* that goes around and around `12` times with the variable `month` counting up from `1` to `12`; comment out (`%`) the printing (`disp`) line[43] (unless you have grown rather attached to it) and move on.

[43] Note that by commenting out a line rather than completely deleting it, if you want to print out the loop count in the future, all you have to do is to un-comment the line, rather than type in the command all over again. This can be really useful if your debug command is long, or particularly if you have a whole series of lines that are required to report the information you want to know.

We can construct filenames to load in by:

1. Forming a complete filename by *concatenating* sperate strings. For example:

   ```
   » filename = ['temp' '1' '.tsv']
   ```

   will create the filename out of 3 components parts – a common elements of all the filenames (`'temp'`), the number of the month (`'1'`), and the file extension (`'.tsv'`).
2. Converting a number value of a (count) variable to a string (the `num2str` function), so instead of hard-coding in the string representing a number (`'1' in thsi example`), you convert from the value of a counter, e.g. `num2str(month)`.

This is where the role of the loop counter (stored in the variable `month`) comes in. Each time around the loop, the value of variable `month` is the number of the month. All you have to do is to convert

this value to a *string* and thereby automatically generate the correct month's filename each time (as per above).

Now add the following within the *loop* in your script;

```
filename = ['temp' num2str(month) '.tsv'];
```

and after it (still within the *loop*) some debugging[44]:

[44] Or you can make use of a **breakpoint**.

```
disp(filename)
```

just to confirm that appropriate filenames are being generated. Save and run the *script*. Satisfy yourself that you know what it is doing. Can you see that you are now automatically generating all the 12 filenames in sequence? And this only takes 3 lines of code total (not including the debugging line), compared with 12 lines if you had to write down all the 12 file names long-hand.

[45] Remember that the load line goes inside the loop. (Why? Try writing it outside the loop (at the end) and see what happens if you like.)

Now *comment out* (or delete) the `disp(filename)` line, and add a new line to load in each dataset from the filename that is constructed each time the loop goes around.[45] This is almost identical to the use of the load command earlier in the example:

```
temp(:,:,month) = load(filename);
```

but ... with 2 important differences. Firstly, rather than specifying the 1st, then 2nd, etc layer of the 3D array, are are specifying the layer with an index equal to `month`, which remember, counts up from 1 to 12 in the *loop*. Secondly, rather than specifying the filename explicitly in the load command, we are passing the string contained in the variable `filename`. Hopefully on the previous line of code within the loop, you have created the string value of `filename` ...

Assign the 2D data array that is loaded in, to the `temp` array variable, at the next layer number. Take a look at the Workspace window – note that you have an array (`temp`) that has size 94×192×12. If temp is 94×192×1 then go back a page or so and go through the bit about loading data into a 3D array. You want to avoid over-writing the information that is already there, so the line; `temp = load(filename);` will not work (and you will only get a 94×92 array after going 12 times around the *loop*). Why? (Again, look back a page-ish.)[46]

[46] If you are still stuck, then stick up a paw.

At the end of (but still within) the *loop* (i.e., before the *loop* has completely finished), create a new figure window on one line, then plot (using `pcolor`) the monthly temperature data on the next line, and add the essential labelling stuff (lines after that). All within the loop still. This line should look something like:

```
pcolor(temp(:,:,month));
```

and should produce extremely exciting graphics as in Figure 2.8[47]. (Don't just type this line in blindly (maybe it doesn't work anyway).

[47] The 2D graphics will get *much* better later – one thing at a time!

Make sure that you understand what you are doing (otherwise why do GEO111 at all?).)

Save and run the *script*. Do you have 12 different temperature plots on the computer screen?[48] Note that if you keep running the program, you'll get 12 more figure windows each time. This is where the `close all` command comes in useful, and you could add this at the start (or end) of your *script*. Because if you re-run the *script*, you wont then end up with 24 figure windows. And then 36 the time after that, and ...

Actually, there is no need to create a new figure window each time – comment out the command that creates a new figure window (`figure`). Save and re-run and note the difference.

Finally ... look up **MATLAB** `help` on `getframe`. Then go back to your global temperature loading/plotting script and add the following line[49]:

```
M(month)=getframe;
```

Save and run. When **MATLAB** is all done, at the commend line type in:

```
» movie(M,5,2)
```

and hopefully ... an animation of the progression of monthly surface air temperatures globally, should appear[50].

If you want to play some more, just type `help movie` – there are controls for not only the number of times you loop through the complete animation, but also for the numbers of frames per second. But we will revisit this later – the 2D plotting you have done so far is *very* basic and there is no scale or sane x/y axes. Later we can also add the continental outlines that will help orient you and improve the quality of the graphical output.

Before you move – go look at your *script* – is it well commented? Would you be able to tell exactly what it does it by the end of GEO111? What about next year? Are the *loop* contents indented? It is important that it is commented and laid out adequately.

Finally – there are **MATLAB** commands to turn your **MATLAB** format movie into a format you can use elsewhere. The old ... and maybe still working (or not) command was `movie2avi` (see Box). The new/replacement command is `VideoWriter`. TO use this new **MATLAB** *function*, adjust your code as follows and as per the **MATLAB** `help` on `VideoWriter`. We could also simplify things by not creating a 3D array, but rather over-writing a 2D array each time (having first created the animation frame).

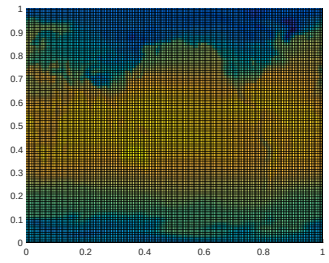

Figure 2.8: Extremely unappealing blocky plot of Earth surface temperature (who cares with month? – the graphics are too poor to matter ...).

[48] If not, stick you paw up in the air for help ...

[49] Where to put the line? See the `Example` given in the `help` on this function. It is exactly what you are doing here.

[50] Note that the active Figure window may have disappeared behind some other windows so go rescue it to see what is happening.

movie2avi

The function `movie2avi` converts an animation encoded in **MATLAB**'s `movie` format to an **avi** file, which is a common film format that can then be played in **Windows** (or other operating systems) without having to use **MATLAB** to display it. It is also a format that could e.g. be embedded in a **Powerpoint** presentation. A typical basic usage is:

```
» movie2avi(M,'file.avi');
```

where `file.avi` is the output filename and `M` the input **MATLAB** `movie` name.

```
% Prepare the new file.
vidObj = VideoWriter('my_animation.avi');
open(vidObj);
% Create an animation.
for month=1:12
   filename = ['temp' num2str(month) '.tsv'];
   temp = load(filename);
   pcolor(temp);
   % Write each frame to the file.
   currFrame = getframe;
   writeVideo(vidObj,currFrame);
end
% Close the file.
close(vidObj);
```

2.5 *Loops and conditionals ... together(!)*

No surprise that you might combine both *loops* and *conditionals* in the same programming structure. In fact, this becomes very powerful and is an extremely common device in programming.

2.5.1 `for ...` and conditionals

As an alternative to (or as well as) a fixed *loop*, or variable and (*function*) parameter passed controlled *loop*, we could specify a near infinite *loop*, but provide a get out of jail free. For example, within the *loop*, we could add a line that asks an additional question: 'Another input (y/n)?' We would test the answer and if no ('n'), exit the *loop* (and report the sum as before). This would look like:

```
my_question1 = 'Please enter a number:  ';
my_question2 = 'Another input (y/n)?  ';
for n = 1:1000000
   my_number = input(my_question1);
   my_string = input(my_question2,'s');
   if strcmp(my_string,'n')
       break
   end
end
```

where`1000000` is simply chosen as a 'very large number' and one rather larger than the maximum number of numbers you could ever imagine entering[51].

The key new command here is `break`. The way the code works (hopefully!) is that at the start of a new iteration of the *loop*, the 'another input' question is asked – if no further input is required, the loop exits via the `break` command. Otherwise (the `else`), the user is prompted for another input. Note that now we have loops and conditionals nested together, it helps even more to *indent* the code[52]. Also note that here – the two different questions (demands) outputted to the screen – 'Another input (y/n)?' and 'Please enter a number' – are pre-defined before the loop starts. These same lines could be placed within the loop, but re-defining the variable e.g. `my_question1` as 'Another input (y/n)?', each and every time, is redundant (i.e. it could instead simply be defined once at the start of the program). Also also note that in this code, the number entered in is assigned to the variable `my_number` rather than `n` as was used before – simply to help distinguish the number input from the string input (assigned to my_string).

It is up to you to 'do' (i.e. add or modify the code) something with the number entered in an stored in the variable `my_number`, as each

`break`
Simply – `break` terminates the execution of a for or while loop'. And from **help** a further clarification: 'Statements in the loop after the break statement do not execute.'
Slightly more complicated (but not much) in the case of nested loops – in this case, `break` exits only the loop in which it occurs.

Indenting code
Just do it (or let MATLAB do it). Even for a single loop or conditional, it is way easier to see what code is within the loop and what outside it, when the code inside starts several spaces in from the margin.
For nested loops and conditionals, it is even more important to keep (visual) track on what is going on.
Note that the indention (or lack of) does not affect the execution of the code (unlike in e.g. Python).

[51] There us a better way of doing this, with the `while` construct, that we'll see shortly.

[52] **MATLAB** will do this for you if you click on the Indent icon. It will also indent the code as far as it reasonably can, as you type.

time around the loop, the previous value is over-written by the new input.

Currently, the program only exits upon entering 'n' to the question. Instead, we could have it exiting for any answer other than 'y':

```
my_question1 = 'Please enter a number:  ';
my_question2 = 'Another input (y/n)?  ';
for n = 1:1000000
   my_number = input(my_question1);
   my_string = input(my_question2,'s');
   if ~strcmp(my_string, 'y')
       break
   end
end
```

which compares `my_answer` and 'y', if this is not true (that they are the same), `break` is executed.

A MORE PRACTICAL EXAMPLE would be when saving a data file, to test for a filename already that already exists and if so, automatically modify the new file name so as not to over-write the file.[53] The relevant function is `exist`, and in the case of a test for a file, returns either `0` (the file does not exist in the **MATLAB** search path, although that does not rule out it existing somewhere else entirely), or `2` (the file exists).

Clearly(?), in the example of saving the movie file (using the `movie2avi` command), you might well want to test whether the filename that you have chosen already exists (i.e. the value returned by `exist` is 2). If so (i.e. the file exists), you need to modify the filename by means of a new concatenation, perhaps appending something like '`_NEW`' to the end of the string[54]. If not, and the filename has not already been used, you can proceed as before – the equivalent of 'doing nothing'. Go ahead – try it (i.e. modify your code to avoid over-writing an existing filename).

You could start by defining a default filename in the code[55] that you will use if there is no clash with any existing file, e.g.

```
my_filename = 'GEO111_movie.avi'
```

Now test whether this filename already exists:

```
filename_check = exist(my_filename,'file')
```

Finally, using an `if` statement, test whether the value of `filename_check` is equal to 2. If so, you are going to need to modify the filename string (`my_filename`). If not, you can let the *conditional* just `end` and proceed to saving. Modifying the filename is just as per for the example of loading global temperature distributions, e.g.

[53] Note that while in the `m-file` Editor, **MATLAB** asks you if you want to overwrite an existing file, when saving a file directly from a program, no such dialogue box or warning is given.

[54] Recall that in using the `movie2avi` command, you pass a filename – simply modify the filename passed, in a similar way to in which you modified the filename for loading the temperature data.

`exist`
Tests for whether a specified variable, function, file, or directory exists, and in generally, which is these it is.
The general syntax and usage is:

```
exist('A')
```

to return what A is.
An extended syntax with a second passed parameter:

```
exist('A','file')
```

returns value of 2 is returned is `A` if a file, and for:

```
exist('A','dir')
```

returns a value of 7 is returned is `A` if a directory.

[55] Either near the very start of the program (neater), or just before you need to use the string (to save a file).

```
my_filename = ['NEW_' my_filename];
```

where here, we take the string contained in `my_filename`, we append a 'NEW_' to the start[56], and assign the new (longer) string back into the variable `my_filename`.

The file naming becomes a little awkward, so rather than the entire filename + extension, you might just store just the filename in the (`my_filename`) variable. i.e.

```
my_filename = 'GEO111_movie'
```

but the remembering when you test fo rthe existence of a particular file, you must add the extension, i.e.

```
filename_check = exist([my_filename '.avi'],'file')
```

(here we create a new string `[my_filename '.avi']` by concatenating my_filename with the extension '.avi'). If the filename exists, the new filename we generate can then be:

```
my_filename = [my_filename '_NEW'];
```

(adding the '_NEW' after, rather than before the existing filename string).

[56] Note that because the filename already has its .avi extension attached, you'll have to modify the start of the string.

2.5.2 while ...

We can re-frame the earlier example programs using the `while` construct rather than the `for` loop. But now ... you need to specify under what conditions the loop continues as the basic syntax (see earlier or **help**) is:

```
while STATEMENT (IS TRUE)
   CODE
end
```

Here – `STATEMENT (IS TRUE)` is the conditional. For instance and rather trivially, create the following as a program and run it[57]:

```
while true
   disp('sucker')
end
```

[57] You ... are going to need a **Ctrl-C** on this one ...

What has happened is that true is always ... true. Hence the condition is always met and the while loop loops forever. Conversely, `while false` would never loop, not even once. more interesting and useful is when the statement might change in value as the loop progresses.

Consider (and type up in a script):

```
n = 0;
while (n < 10)
   disp('sucker')
end
```

This also will loop for ever as `n` is initialized to `0` and hence the statement `(n < 10)` is always true. But if we increment the value of `n` each time around the loop:

```
n = 0;
while (n < 10)
   disp('not a sucker')
   n = n + 1;
end
```

then the loop will execute exactly 10 times (just as per `for n = 1:10`). You could also do this in reverse:

```
n = 10;
while (n > 0)
   disp('not a sucker')
   n = n - 1;
end
```

Now, `n` counts down from `10` and when it reaches a value of `0`, it is no longer greater than zero and the statement (`n > 0`) is false (and the loop terminates).

It is not always completely obvious whether even simple while loops like this execute 9 or 10 (or 11) times particularly when often you might come across `while (n >= 0)` that allows the loop to continue when when `n` has reached z value of zero (but not below). So – spend a little while playing about with different `while` configurations and loop criteria.

Finally, note that the conditional statement in the while loop need not test for an integer being larger or smaller than some threshold. One could equally loop on the basis of a string equality/inequality. For example, taking the previous example using `break` could be recoded with a `while` loop:

```
my_question1 = 'Please enter a number:  ';
my_question2 = 'Another input (y/n)?  ';
my_string = 'y';
while strcmp(my_string,'y')
   my_number = input(my_question1);
   my_string = input(my_question2,'s');
end
```

and ends up a slightly shorter and more compact piece of code, omitting the need for a `break` or a nested structure. However, in this example, we do need to initialize the value of `my_string` (to 'y' – assuming that we want at least one number). Try it and then adjust it so that the loop proceeds as long as the answer is not 'n' (rather than as long as it is 'y')[58]. Note that as before – it is up to you to 'do' (i.e. add or modify the code) something with the number entered in an

[58] See earlier Example.

stored in the variable `my_number`, as each time around the loop, the previous value is over-written by the new input.

Extending the filename checking example ....[59] to fully integrate a *loop* and *conditional*. The problem with the previous code is that you checked for the existence only a default filename (and appended '_NEW' if a file already existed).

One (partial) solution would have been, rather than append a predefined string ('_NEW') to the filename, would be to request that the user provide either a string to append, or a completely new filename. You have already see the `input` command in action, so you should be in a good position to code this modification up.[60]

A better solution (because even when asking for an alternative filename – what if that file exists too?) would be to keep checking for a filename clash and keep asking for a new filename, until a unique one is found. Who knows how many attempts this might take (to find an unused filename), so `while ...` would be a better choice of loop than `for ...`. Becasue `exist` returns a `2` if the file already exists, a logical condition for `while` would be while `exist` is returning `2`:

```
my_question = 'Please enter an alternative filename (without
the extension):  ';
while (filename_check == 2)
   my_filename = input(my_question,'s');
   filename_check = exist([my_filename '.avi'],'file')
end
```

Within the loop, a new filename is requested and then checked against the directory contents. What is missing is the initial value of `filename_check`. In a previous example, we simply set a value at the start. If we did that here, the first line of this code would look like:

```
filename_check = 2
```

In this case, we do not need a default filename as the user provides the very first filename that is tested. Alternatively, we could perform a single check before the *loop* starts:

```
my_question = 'Please enter an alternative filename ...
   (without the extension):  ';
my_filename = 'GEO111_movie';
filename_check = exist([my_filename '.avi'],'file')
while (filename_check == 2)
   my_filename = input(my_question,'s');
   filename_check = exist([my_filename '.avi'],'file')
end
```

[59] Which first time around did not actually combine loops and conditionals in the same structure. Rather, a loop came first in the program (loading in and plotting the temperature data), ended, and only then a conditional checking the filename.

[60] Effectively, all you have to do, if `exist` returns a `2` and the file already exists, is to ask for an alternative filename, and use the string entered in as the new filename (and don't forget to add the '.avi' extension to the end when saving)

2.6 *Even more (and loopier) loops*

[Further examples of increasingly extreme loopiness.]

Looping through arrays. In plotting e.g. global temperature distributions, it would be nice to add on the continental outline. Currently, and particularly with the very basic 2D plotting you have seen so far (`pcolor`), you are left to some extent guessing where the land and where the ocean is.

A pair of files are provided (from the website), comprising a series of pairs of lon-lat values that delineate the outline of the continents and all but the smallest of islands:

```
continental_outline_lat.dat
continental_outline_lon.dat
```

Download, and load these into the **MATLAB** workspace (in the 'usual way'). You should now have 2 vectors. Maybe view then in the Variable Window to get a better idea of what you are dealing with. Also keep an eye on the entries in the Workspace Window and perhaps the Min and Max values to give you an idea of the range (here: of longitude an latitude values).

Try plotting these lon/lat locations. Use the `scatter` plotting function (which makes it all the easier as your data is in the form of 2 vectors already). You might need to reduce the size of the plotted points (refer to the earlier exercises, or `help`) and additionally, you might want to fill the points (up to you). Remember you can set the axis limits, which presumably should be 0 to 360 or -180 to 180, on the x-axis (longitude), and -90 to +90 on the y-axis (latitude). Font sizes of labels can also be increased if necessary. You might end up with something like Figure 2.9.

By plotting dots (points), the coastal outline at higher latitudes gets increasingly pixelated (why?). So, we might instead plot as lines between the lon-lat pairs. For this, you could simply use `plot`. Do this, and see if you get something like Figure 2.10..

Well ... interesting. If you think about it, as one continental outline is completed, the next lon-lat pair will be for the next continent or island. What `plot` does is to join up *all* the adjacent x-y (lon-lat) pairs and hence points, which is why you get the straight lines criss-crossing the map with the start of each successive continent and island in the dataset joined to the end of the previous one.

The continental outline dataset is not actually that useless. There are additional files that specify which block of lon-lat pairs belong to a single shape (i.e. continent or island). Load in the 2 additional files:

```
continental_outline_start.dat
```

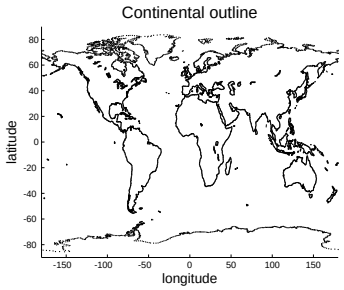

Figure 2.9: Continental outline (of sorts).

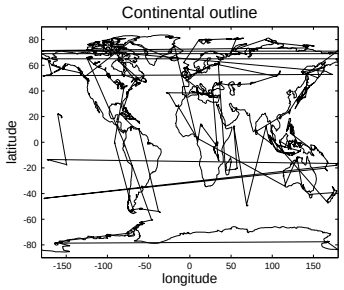

Figure 2.10: Another continental outline (of sorts).

```
continental_outline_end.dat
```

These vectors hold information regarding the start row and end row, of each shape. Again, view the contents of these vectors to get an idea of what is going on. For example, you'll see that the first entry is that the first shape starts on row `1` (`continental_outline_start.dat`), and ends on row `100` (`continental_outline_end.dat`). The 2nd shape starts on row `101`, and ends on row `200`. etc etc

The simplest way too start dealing with all this, is to just plot the very first shape, defined by rows `1-100` of the lon and lat vectors. By now, you hopefully will be able to see that to plot rows `1-100` of lon and lat data, you are going to do:

```
plot(lon(1:100),lat(1:100));
```

(here I have named the arrays `lon` and `lat` for added convenience rather than the long-winded default file-name based versions (`continental_outline_lat`, `continental_outline_lon`)).

Well ... this is probably about as unexciting as it gets – a small piece of the Antarctic coastline. If you do a `hold on` and plot the next block (rows `101-200`), you'll get the next chunk of coastline. (Try this and see.) You could keep going this – manually adding additional sections of the global continental outline. This could get tedious ... and it turns out that there are 283 different fragments to plot, all one after another. (This number comes from asking **MATLAB** the `length` of `continental_outline_start.dat` or `continental_outline_end.dat`.) This is, of course, why we need to get clever with a *loop* and automatically go through all 283 fragments, plotting them on on top of another in the same figure.

How? First you need to write the `plot` command in a more general form – you do not want to have to read the values out of the `continental_outline_start.dat` and `continental_outline_end.dat` files manually. Hopefully, it should be apparent that you can re-write the plot statement for the first fragment, as:

```
plot(lon(line_start:line_end),lat(line_start:line_end));
```

where for the first fragment, the values of `line_start` and `line_end` are given by `lstart(1)` and `lend(1)`, respectively (renaming the original vectors to shorten the variable name)[61]. Re-writing again:

```
plot(lon(lstart(1):lend(1)),lat(lstart(1):lend(1)));
```

Try this and check you still get the single piece of the Antarctic coastline.

Really, you should hopefully be making the mental leap to looking at `(1)` and thinking that it could be: `(n)`, where `n` is a loop counter which can go from 1 to 283[62] and hence loop through all the line

`length`
This function could almost not be simpler – just pass the name of a vector, and it returns its length (i.e. the number of rows, or columns, depending on the shape of the vector).

[61] You cannot use the obvious variable name `end` – why not?

[62] This number comes from a 5th file – `continental_outline_k.dat`, that numbers the continents/islands from 1 to 283. You don't need it, although downloading it, loading it, and determining the length f the vector gives you the loop limit and you would not have to go trusting me to write down 283 correctly without making a mistake ...

fragments. Yes? For instance, setting `n=1`, and plot (with `n` replacing `1` in the code fragment above) – you should again get that very first fragment. Try setting `n=283` and plot. Do you get the last fragment (what is it of[63])?

So ... create yourself an `m-file`. Load in the lon-lat pairs as vectors (renaming then to something more manageable if you wish). Load in the vectors continuing the start and end information. Create a `do ... end` loop. Maybe print (`disp`) the loop count and run the program (after saving), just to check first that the loop is functioning correctly. Before the loop, create a Figure window. and set `hold on`. You now have a basic shall of a program – loading in the data, initializing a figure, and appropriate looping, but not yet actually doing anything within the loop.

In the *loop* all you need is the `plot` command, but with the start and end rows being a function of `n` (or whatever you call the loop counter). Set axis dimensions and label nicely (after the loop ends). Run it. Hopefully ... something like Figure 2.11 appears(?)

[63] An island at about 20N and -150E if you have done it correctly.

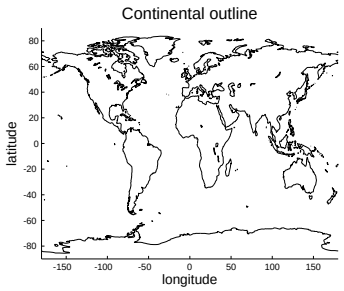

Figure 2.11: Another go at the continental outline!