ANDY RIDGWELL

STR=’DO YOU LIKE BANANAS?’ [EXAM VERSION]

UNIVERSITY OF CALIFORNIA, RIVERSIDE / DEPARTMENT OF EARTH SCIENCES
2015/6

Copyright © 2016 Andy Ridgwell

http://www.seao2.info/teaching.html

Except where otherwise noted, content of this document is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 license (CC BY-NC-SA 3.0) (http://creativecommons.org/licenses/by-nc-sa/3.0/)

First printing, December 2016

Contents

1 Elements of ... MATLAB and data visualization 13

1.1 Using the MATLAB software 14

1.1.1 Starting MATLAB 14

1.1.2 The command line 14

1.1.3 MATLAB GUI 14

1.1.4 Help(!) 15

1.2 Basic concepts 16

1.2.1 Variables 16

1.2.2 Numerical expressions 18

1.2.3 Relational and logical operators 19

1.2.4 Functions (built-in) 20

1.2.5 Miscellaneous commands 20

1.3 Vectors and arrays #1 22

1.3.1 Creating vectors 22

1.3.2 Basic vector manipulation 22

1.3.3 Addressing elements in vectors 23

1.4 Basic graphing (aka. 'data visualization') 25

1.4.1 Plotting 25

1.4.2 Graph labelling 26

1.4.3 Sub-plots 26

1.4.4 Saving graphics and figures 27

1.5 Vectors and arrays #2 28

1.5.1 Creating matrices and arrays 28

1.5.2 Basic matrix manipulation 29

1.5.3 Some matrix math :(31

1.6 *Loading and saving data* 32
1.6.1 *Where am I?* 32
1.6.2 *Loading and importing data* 33
1.6.3 *Saving and exporting data* 33
1.6.4 *Loading and saving the workspace* 33
1.7 *Basic data processing* 34
1.8 *Yet more graphing* 38
1.8.1 *Modifying lines/symbols in* `plot` 38
1.8.2 *Plotting multiple data-sets* 38
1.8.3 *Scatter plots* 39
1.8.4 *Histograms* 41
1.8.5 *Simple 2D data and bitmap visualization* 42

2 *Elements of ... programming* 43
2.1 *Introduction to scripting (programming!) in MATLAB* 44
2.1.1 *Programming good practice* 45
2.1.2 *Debugging the bugs in buggy code* 47
2.2 *Functions* 50
2.3 *Conditionals '101'* 52
2.3.1 `if ...` 52
2.3.2 `switch ...` 56
2.4 *Loops '101'* 58
2.4.1 `for ...` 58
2.4.2 *Other loop configurations and usages* 61
2.4.3 *Fun(!) worked examples* 62
2.5 *Loops and conditionals ... together(!)* 66
2.5.1 `for ...` *and conditionals* 66
2.5.2 `while ...` 68
2.6 *Even more (and loopier) loops* 71

3 *Further ... MATLAB and data visualization* 75

3.1 *Further data input* 76

3.1.1 *Formatted text (ASCII) input* 76

3.1.2 *Importing ... Excel spreadsheets* 79

3.1.3 *Importing ... netCDF format data* 80

3.2 *Further (spatial / (x,y,z)) plotting* 84

3.2.1 *Plotting maps* 91

4 *Further ... Programming* 93

4.1 `find!` 94

5 *Graphical User Interfaces (GUIs)* 101

5.1 ***MATLAB** GUI basics* 102

5.1.1 *Hello, World [Static Text (box)]* 103

5.1.2 *Simple GUI responses [Push Button]* 105

5.1.3 *Updating object properties (do you like bananas?)* 106

5.2 *GUI Pokemon game* 110

6 *zero-D / equilibrium modelling* 123

6.1 *A zero-D Energy-balance model of the climate system* 124

6.1.1 *The basic EBM* 125

6.1.2 *The EBM as a function* 126

6.1.3 *Parameter sensitivity experiments using the EBM – #1* 126

6.1.4 *Parameter sensitivity experiments using the EBM – #2* 128

6.1.5 *Creating a function for the evolution of solar constant through geological time* 131

6.1.6 *Using multiple functions and calculating global surface temperature as a function of geological time* 132

6.2 *'Daisy World'* 133

6.2.1 *'fixed daisy' daisy-world* 134

6.2.2 *'dumb daisy' daisy-world* 135

6.2.3 *'clever daisy' daisy-world* 138

7 *Dynamic (time-stepping) modelling* 141

7.1 *Catch the ball (ballistics and simulating trajectories)* 142

7.2 *Dynamics in the zero-D Energy-balance climate model* 149

Bibliography 153

Index 155

List of Figures

1.1 Default output of `plot`. 25
1.2 A plot illustrating axis auto-scaling (maximum x and y values now slightly larger than 10 and 100, respectively). 26
1.3 A (only very slightly) improved plot. 26
1.4 Arrangement of subplots. 27
1.5 Spline fit to measured changes in CO_2 concentration in Law Done ice core, following *Etheridge et al.* [1996]. 33
1.6 proxy reconstructed past variability in atmospheric CO_2. 34
1.7 Proxy reconstructed past variability in atmospheric CO_2 (sorted data). 35
1.8 Proxy reconstructed past variability in atmospheric CO_2 (sorted data). 38
1.9 Proxy reconstructed past variability in atmospheric CO_2 (sorted data). 39
1.10 Proxy reconstructed past variability in atmospheric CO_2 (scatter plot). 39
1.11 Proxy reconstructed past variability in atmospheric CO_2 (scatter plot). 39
1.12 A 2D plot of some random gridded model data. 42
1.13 A 2D plot of some random gridded model data ... but with the underlying data matrix re-orientated before plotting. 42

2.1 Output from the (bug-fixed version of) `plot_some_dull_stuff` **m-file**. 49
2.2 Extremely unappealing blocky plot of Earth surface temperature (who cares with month? – the graphics are too poor to matter ...). 65
2.3 Continental outline (of sorts). 71
2.4 Another continental outline (of sorts). 71
2.5 Another go at the continental outline! 73

3.1 Very basic imaging (`image`) of an array (2D) of data – here, global bathymetry. 85
3.2 Slightly improved very basic imaging (`imagesc`) of bathymetry data. 85
3.3 Example result of basic usage of the `contour` function. 87
3.4 Example usage of `contourf`, with the `hot` *colormap* (giving dark/brown colors as deep ocean, and light/white as high altitude). 87
3.5 Example usage of `contour`, contouring only the zero height isoline, and providing a label. 87

3.6 Usage of `contour` but with lon/lat values created by `meshgrid` function and passed in (and with the `hot` *colormap* (giving dark/brown colors as deep ocean, and light/white as high altitude). 89
3.7 Example contour plot including `meshgrid`-generated lon/lat values. Result of `contourf(lon,lat,temp7,30)`, where the data file was `temp7.tsv`, with some embellishments. 91

4.1 Proxy reconstructed past variability in atmospheric CO2 (scatter plot). 98
4.2 Proxy reconstructed past variability in atmospheric CO2 (scatter plot). 99

5.1 Starting GUI window of the **MATLAB GUIDE**, GUI design tool. 102
5.2 (Blank) GUI window editor GUI window. 103
5.3 Design of the Hello, World window! 104
5.4 Design window with a default push button object. 105
5.5 (completely) Bananas design window. 106
5.6 (completely) Bananas GUI in action. 108
5.7 Screen-shot of he Pokemon game App. 110
5.8 Template App with background image. 116
5.9 Template App with background image plus Pokemon. 116
5.10 Template App with background image plus small Pokemon at bottom right. 117
5.11 Template App with background image plus small Pokemon at bottom right, now with its transparency applied. 117
5.12 App with ball trajectory trail. 119

6.1 Form of the basic EBM model. 125
6.2 Form of the basic EBM model as a *function*. 125
6.3 Schematic structure of the model configured to carry out a single parameter sensitivity study. 128
6.4 Sensitivity of global mean surface temperature vs. solar constant (mean surface albedo held constant at an albedo value of 0.3). 128
6.5 Chess board grid pattern. 130
6.6 Schematic structure of the model configured to carry out a double (in terms of solar constant AND now albedo) parameter sensitivity study. 130
6.7 Global mean surface temperature (°C) as a function of solar constant and surface albedo grid point number. 130
6.8 Global mean surface temperature (°C) as a function of the value of solar constant and surface albedo. 130
6.9 Schematic structure of code for calculating the solar constant (output) as a function of time (input). 131
6.10 Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, and solar constant and EBM functions. 132

6.11 Simple EBM projection of the evolution of Earth surface temperature with time. Time at the present-day is highlighted by a vertical line (drawn using the **MATLAB** `line` function). 132
6.12 Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, the solar constant and EBM functions, and now the 'daisy' albedo function. 134
6.13 Evolution of global surface temperature and the two populations of daisies with time ... but with no change allowed in the daisy populations (d'uh!). The fractional coverage of white daisies is shown by large empty circles, and for black, by small filled black circles. Data points for mean surface temperature are color-coded by temperature (color scale not shown). 135
6.14 Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, the solar constant, EBM, and 'daisy' albedo functions. Note the creation of an inner loop, with EBM, and 'daisy' albedo functions called from within this, while the solar constant remains called form the start of the outer loop as before. 136
6.15 Evolution of global surface temperature and the two populations of daisies with time ... but now assuming that the growth of each depends on the global mean surface temperature. 138
6.16 Evolution of global surface temperature and the two populations of daisies with time. 139

7.1 Schematic of the thrown-ball system. 142
7.2 Schematic of the code for simulating the horizontal movement of a ball. 142
7.3 Schematic of the code for simulating the horizontal movement of a ball. 144
7.4 Trajectory of a ball!! 148
7.5 Schematic of the script for the basic dynamic EBM 149
7.6 100 yr spin-up of the basic EBM. 149
7.7 Schematic of the script for the basic dynamic EBM – now with added loop count(!) 150
7.8 100 yr spin-up of the basic EBM, but with a poor choice of time-step ... 150
7.9 Schematic of the dynamic EBM as a function and with the CO_2 concentration passed in. 151
7.10 Schematic of the dynamic EBM driven by a history of CO_2 (read in from a file). 152
7.11 Transient EBM response to observed changes in atmospheric CO_2. For reference, the pre-industrial equilibrium global temperature is shown as a horizontal black line. 152

7.12 Transient EBM response to (fake) changes in atmospheric CO2. 152

List of Tables

1
Elements of ... MATLAB and data visualization

Hello Newbies! This first lab's porpoise is to start to get you familiar with what MATLAB is all about and understand how to import and manipulate (array) data in this software environment and do some basic plotted (aka 'data visualization'). If your are clever, you might find menu items or buttons to click that will do the same thing as typing in boring commands at the command line. In fact, you would have to be pretty dumb not to notice all that brightly colored eye-candy in the GUI (Graphical User Interface – i.e., menus, buttons, and stuff) at the top of the screen. However, you will get to grips with programming much quicker if you stick with the instructions and do almost everything that is asked of you using the command line (rather than doing stuff via the GUI), at least to start with. You'll just have to trust me for now ... We'll start with the very basics and things that you could easily do in Excel instead, and build up.

Graphics is one of the important strengths of **MATLAB**. Although other software packages and scripting languages exist that perhaps have the edge on **MATLAB** in terms of visually appealing plots and graphs, **MATLAB** is worlds apart from e.g. **Excel**.

1.1 Using the MATLAB software

1.1.1 Starting MATLAB

To start with: find the MATLAB icon on the desktop; run the program. You should see a number of sub-windows arranged within the main MATLAB window, hopefully including at the very least, the *Command Window*[1]. Depending on whether you have used MATLAB before and it has remembered your settings, windows may also include: *Command History*, *Workspace*, *Current Folder*. If instead you see; 'Tetris', 'Grand Theft Auto: San Andreas', and 'World Championship Pool', then you have the wrong software running and are going to find learning MATLAB rather hard. However, there is big $$$ to be made in on-line gaming tournaments these days. Would you really rather be a geologist and spend the rest of your days hitting rocks with a hammer? If so, read on ...

[1] Conveniently labelled `Command Window` – you cannot possibly fail to identify it ...

1.1.2 The command line

When MATLAB initially starts up, the *Command Window* should display the following text:

```
Academic License
»
```

or in order versions of the software:

```
To get started, select MATLAB Help or Demos from the Help
menu.
»
```

but in either case, with a vertical blinking line (cursor) following the double 'greater than' symbols[2].

If you are unfamiliar with using command-line driven software ... Don't Panic! Nothing bad can happen, regardless of what you do. Well, almost. It is possible to accidently clear MATLAB's memory of the results of calculations and data processing and close plots and graphs before you have saved them, but MATLAB remembers all the commands you type, so in theory it is perfectly possible to quickly reproduce anything lost. (Later on we will be placing the sequence of commands into a file (that is saved) and so ultimately, MATLAB should turn out to be mostly fool-proof.)

[2] Note that in nerd-speak the » is called the command 'prompt' and is prompting you to type some input (Commands, swear words, etc.). See – the computer is just sat there waiting for you to command it to go do something (stupid?). If one does not appear at the bottom of whatever is in the *Command Window* is means that MATLAB is busy doing something extremely important. Or perhaps, MATLAB may have completely died. Either way, it will not accept any new/further commands until it is done calculating/dying.

1.1.3 MATLAB GUI

There are lots of fancy looking icons and pretty colors and you could spent all day staring at them and not getting any work done. Or

learn good programming practice. Which is why we mostly will ignore the eye-candy and little (if any) guidance will be given as to the functionality of the GUI. Look at this as a lesson for the user (to read the Help, textbook, on-line documentation, or simple go Google for an answer[3]).

[3] i.e. Internet fishing

1.1.4 Help(!)

Press `F1` or click on the question mark icon on the tool-bar, to bring up the indexed and searchable MATLAB documentation.[4]

[4] It is also possible to obtain context-specific help, e.g. on a specific (built-in) *function*, which we'll see in due course.

1.2 Basic concepts

1.2.1 Variables

A *variable* is, in a sense, a pointer to a location in computer memory where a piece of information is stored[5]. A variable is associated a name to make things rather more easy and convenient. The name can be anything you like in MATLAB, as long as it does not contain numbers or special characters. So actually, you are only allowed sequences of letters (otherwise knows as 'words'). But you can create a variable name based on 2 (or more) words, separated by an underscore (_). Valid variable names would include:

```
A
B
cat
derpyhooves
this_is_boring_stuff
BIG
big
```
[6]

[5] In the bad old days, this pointer was the actual address in memory and might have looked something like `f04da105`.

[6] Note that MATLAB distinguishes between lower and UPPER case letters in a variable (i.e. `BIG` and `big` would represent two different and distinct variables).

Variables are entirely useless unless they have some information assigned to them. In fact, you can type in any of the variable names above (at the command line) and MATLAB will deny it knows what you are talking about[7].

[7] Technically, MATLAB reports: `Undefined function or variable` which tells you it is neither a function name (more on this later), nor is defined as having any information associated with it.

So far so useless – you need to *assign* something to it. Which brings us to quite 'what' and 'how'. First of, you need to know that variables can have the following *types*:

- **Integer** – An integer number is a counting number, i.e. `1, 2, 3, ...` and including zero and negative integers. MATLAB has different representations for integer numbers, depending on how large a number you need to represent (and how much memory it will need to allocated to storing it). This is something of a throwback to the days when computers only had $1/10000000^{th}$ of the memory of your iPhone and were slower than a lemon.
- **Real (floating point)**[8] – A real number can have a non-integer component, e.g. 1.5 or $6.022140857 \times 10^{23}$. Real numbers also come in different precisions in MATLAB (also to do with memory allocation and speed), determining not just the number of decimal places that can be represented, but also the maximum size.
- **String (character)** – One or more characters, but now allowing spaces (unlike in the case of naming variables).
- **Logical** – `true` or `false`.
- **etc** – No, not a real type, but to note that MATLAB defines and recognises a whole bunch of other types, including **Complex**

[8] The distinction (sort of) is that *floating point* is a specific representation of a **real** number.

(MATLAB can handle *complex numbers*) and **Object** (we will also not worry about *objects*, which can incorporate a combination of types. At least, not yet ...).

The first thing to learn is to ideally, do not attempt to mix up (combine) variables of different types. MATLAB is very forgiving when it comes to combining an *integer* and a *real* number in the same calculation, but in other programming languages, this should be avoided. However, even in MATLAB, *strings* and *reals* (or *integers*) are very different things. When necessary, different variable types can be converted between (see **Variable Type Conversion** Box).

The second and perhaps rather more important thing, is how to assign a value to a variable (and in fact, create the variable in the first place). Programming languages such as FORTRAN require you to define the variable beforehand and assign it a type. MATLAB allows you to define and assign a value to a variable all at the same time, and it will kindly work out the correct type based on the value you assign to it. You assign a value using the assignment operator `=`[9]. For example:

```
A = 10
```

will assign the value `10` to the variable `A`. If you type this at the command line, MATLAB will kindly repeat what you have just told it and report the value of `A` back to you:

```
A =
10
```

Note that you do not need to add a space before and/or after the assignment operator (`=`). This is something of a personal programming and aesthetics preference, i.e. whether to pad things out with spaces or not. (Chose what you feel happiest with and later on, whatever leads to the fewest programming mistakes ...)

MATLAB will also report in the *Workspace* window, the name and value, type (called *Class*), etc of all your current variables (just one currently?). Actually, it is not all quite so simple. If you take a look at the *Class* of the variable `A` in the display window – it is listed as *double* (a *real* number) rather than an *integer*. So by default, if MATLAB does not know what you really want, it defines `A` as a double precision real number[10].

The next complication comes when assigning a string (a sequence of characters) to a variable. For example, try:

```
B = apple
```

and MATLAB is far from happy. As it turns out, a sequence of characters can also refer to a *function*[11] in MATLAB, and this is what

Variable Type Conversion

MATLAB provides a variety of *functions* (see later) for converting between different *types* of *variables*. The most commonly-used/useful ones are as follows:

1. converting from a number to a *string* (`s`)
   - `s = num2str(N)`, where `N` is any number type variable
   - `s = int2str(I)`, where `I` is an integer
2. converting from a *string* (`s`) to a number
   - `x = str2num(s)`, where `N` is (generally) a double precision (*real*) number

Case #1 (`num2str`) is generally the most useful, e.g. in adding specific captions to plots (with caption text based on the value of a numerical variable) – examples are given later.

[9] This is NOT 'equals' in MATLAB. We will see the *equality operator* shortly. `=` assigns the value or variable on its right the variable on the left.

[10] If you genuinely wanted an integer, there are ways to do this, such as using a type conversion function form *real* to *integer* (see above).

[11] You will see *functions* shortly. For now – note that they are 'special' (reserved) words that perform some action and hence cannot also be used for a variable name.

MATLAB looks for (i.e. a match to `apple` in the list or variable (and function) names). To delineate `apple` as a string, you need to encase it in (single[12]) quotation marks:

```
B = 'apple'
```

Just as MATLAB creates new variables on the fly, you can re-assigned values to an existing variable, even if this means changing the *type*, e.g.

```
A = 'banana'
```

has now replaced the real number `10` with the character string **banana** in variable `A`. This is reflected in the updated variable list details given in the *Workspace* window (and a *Class* now listed as `char`).

Finally, it is possible to suppress output to the *Command Window* when making *assignments* – simply an a semi-colon (`;`) to the end of the *assignment* statement, i.e.

```
C = 'banana';
```

now does not results in anything being echoed to the command line (but the *Workspace* is still updated to reflect this variable assignment). If you wish to see the contents of the variable, you can either just type its name at the command line, or view its value as listed in the *Workspace* window.

[12] Double "" quotation marks will not work.

1.2.2 *Numerical expressions*

You can do normal maths in MATLAB. Or at least, something that looks at least a little intuitive. (In fact, I often use MATLAB as a calculator.) The primary/common numerical expressions are:

- `exponentiation` — `^` — raises one number of variable to the power of a second, e.g. a^b, a to the power b, which is written in MATLAB as `a^b`.
- `multiplication` — $\times$ — e.g. a$\times$b, written in MATLAB as `a*b`.
- `division` — `/` — (written as you would expect).[13]
- `addition` — `+` — (guess).
- `subtraction` — `-` — again, obvious/intuitive.

[13] Entertainingly, it turns out that if you write the reverse, backslash character (\) in the equation, you divide the over way (i.e. denominator divided by numerator).

The order in which the numerical operators are written down is important and MATLAB will execute them in a specific order (operators higher up the list, executed first), i.e. first `^`, then `*,/`, and last `+,-`. There is also 'negation', when you change the sign of a variable, and which is executed immediately after exponentiation. The assignment operator (`=`)[14] comes last. If you are unclear about the order numerical operators are carried out, then place parentheses `()` around the component of the calculation you wish to be carried

[14] This is **NOT** 'equals to', as you'll see shortly.

out first to enforce a particular order (this can also help in making an equation easier to read and ultimately, easier to debug code). For example, consider:

```
A = 3;
B = 6;
C = 2;
D = C*(A/B+1)
E = C*A/(B+1)
F = C*A/B+1
G = A*C/B+1
```

Try these out (and make up your own combinations) and confirm that the answers are what you would expecty them to be.

1.2.3 Relational and logical operators

We will see more of *relational and logical operators* later when we start to get into some proper coding. For now, you only need to know that a relational operator is one of:

- `greater than` — MATLAB symbol >
- `less than` — MATLAB symbol <
- `greater than or equal to` — MATLAB symbol >=
- `less than or equal to` — MATLAB symbol <=
- `equality` — MATLAB symbol ==
- `inequality` — MATLAB symbol ~=

and test the relationship between 2 variables. Note in particular, that the `equality` symbol (that tests the equivalence between two variables) is represented by TWO = characters `(==)`, and remember that a single = character is the `assignment operator`.

In everyday language, the answer to any one of these relational tests would be a 'yes' or a 'no'. But in MATLAB (and other computer languages), the answer is given as the binary (logical) equivalent where 'yes' is represented by `1` and 'no' by `0`. You can also use `true` (`1`) and `false` (`0`), e.g. `A = true` returns:

```
A =
     1
```

Finally, the *logical operators* (again, more on this later) are:

- `or` — symbol ||
- `and` — symbol &&
- `not` — symbol ~

For now – be familiar with how numerical expressions are written in MATLAB (you'll need to be using these from the outset), and keep in mind the existence of *relational and logical operators*.

1.2.4 *Functions (built-in)*

MATLAB provides numerous built-in `functions`[15]. These `functions` have specific names assigned to them, so care needs to be take not to give a variable the same name as a function to avoid getting confused further down the road. Giving an exhaustive list (and brief description) is outside the scope of this document[16]. Common `functions` will be progressively introduced as this text progress. Note that in addition to the on-line Help documentation, information on how to use a function and example uses is provided by typing `help` and then the function name (separated by a space) at the command line.

MATLAB also provides several built-in mathematical constants (saving having to define a variable with the appropriate number). This are simply variables that have been already defined and assigned values, but which you cannot change (hence the term 'constant'). For instance, the value of π, is assigned to a built-in variable with the name `pi`. You can access (display) its value by typing its name at the command line:

```
» pi
ans =
    3.1416
```

In this example, the use of the *function* is rather trivial – you need to tell the `pi` absolutely nothing, and it spits back the same thing (the value of π) each and every time. In most other *functions*, you will find that you have to pass some information, and the return value will depend on the input. (This will all become apparent in due course ...)

1.2.5 *Miscellaneous commands*

Related to what you have seen so far and will see soon, useful miscellaneous commands include:

- `clear` — Removes all variables from the workspace.
- `clear all` — (Removes all information from the workspace.)
- `close` — Closes the current figure window.
- `clear all` — (Closes all figure windows.)
- `exit` — Exits MATLAB and hence enables additional drinking time in the bar.

Note that a useful trick – if you want to re-use a previously used command but don't want to type it in all over again, or want to issue a command very similar to a previously-used one – is to hit the UP arrow key until the command you want appears. This can also be edited (navigate with LEFT and RIGHT arrow keys, and use Delete

[15] We will be constructing our own later, at which point it should become apparent that there is nothing particular special about them.

[16] A full list of functions can be found in the MATLAB Help Documentation under *functions*.

and Backspace to get rid of characters) if needs be. Hit Enter to make it all happen.

1.3 *Vectors and arrays #1*

So far, your variables have all be what are known as *scalars* – i.e. single numbers (or strings). One of the most powerful things about MATLAB is its ability to represent vectors (1D columns or rows of numbers or strings) and arrays – 2D and higher dimensional regular grids of numbers or strings. (*matrix*[17] is the name commonly given to a 2-D array.)

[17] Not to be confused with the film starting Keanu Reeves.

1.3.1 *Creating vectors*

Vectors are 1-D arrangements of numbers (or strings). You can enter them into MATLAB as a list of space-separated value, encased in (square) brackets, `[ ]`, e.g.

```
B = [0.5 1.0 1.5 2.0 2.5]
```

or with the value comma-separated:

```
B = [0.5, 1.0, 1.5, 2.0, 2.5]
```

Either way, you end up with a vector on its side as a single row of numbers which in math-speak would look like:

$$B = \begin{pmatrix} 0.5 & 1.0 & 1.5 & 2.0 & 2.5 \end{pmatrix}$$

You can also create the equivalent, upright orientated vector (as a single column of numbers) by separating the elements by a semi-colon:

```
C = [0.5; 1.0; 1.5; 2.0; 2.5]
```

which gives the maths-speak representation:

$$C = \begin{pmatrix} 0.5 \\ 1.0 \\ 1.5 \\ 2.0 \\ 2.5 \end{pmatrix}$$

The `colon operator` can be used to much more rapidy create vectors (as long as the elements form a simple sequence in value) as compared to typing in the list of values explicitly. There are two variants to the syntax:

```
A = j:k
```

and

```
A = j:i:k
```

In the first example, `j` and `k` and the minimum and maximum values in the sequence of numbers in the vector. MALAB completes the sequence by assuming that the values monotonically increase and that the elements are separated by one (1.0) in value. e.g.

```
» A = 0:3
A =
    0 1 2 3
```

Note that MATLAB is not inclined to let you directly create a vector of elements that decrease in value (you'll need to flip this puppy about to re-order it if that is what you want – see later).

In the second example, `i` is the increment MATLAB will use to complete the sequence from `j` to `k`. In the example in the text, you could have created the array `B` by typing:

```
» B = 0.5:0.5:2.5
B =
    0.5000 1.0000 1.5000
2.0000 2.5000
```

(More commonly, you might place the colon operator and its min/(/increment)/max values inside a pair of brackets, i.e. `A = [0:3].` so that it is unambiguous that you are creating an array

1.3.2 *Basic vector manipulation*

There are several basic and very useful ways of manipulating vectors (and as we'll see later – matrices). To start with, you might want to determine the orientation and length of a vector. There are several different ways to go about this, which in order of grown-up-ness are:

1. Display the contents of the vector in the command window by typing its name at the command line. Obviously, this will quickly become useless for very large vectors[18].

[18] Try creating a vector from 1 to 100,000 and then displaying it ...

2. Refer to the Workspace window, although this also ends up a total Fail for long vectors.
3. Use the `length` or `size` function (see Box).

If you find that you want a different orientation (row vs. column) of the a vector, the vector can be flipped around (converting row-to-column and column-to-row) using the *transpose operator* (.'), e.g.:

```
D = B'
```

will turn the vector `B` into one (assigned to the variable `D`) with he same orientation as `C`.

You can also re-order the values in a vector (hence addressing the restriction in using the colon operator to create a vector that the values must be monotonically increasing rather than decreasing). Depending on the orientation of the vector, you can use either the `flipud` (for column vectors), or `fliplr` (for row vectors), functions to re-order the elements.

1.3.3 *Addressing elements in vectors*

Values can be extracted from a vector by specifying the index (technically, this should be an integer, but MATLAB is pretty forgiving and you can get away with using a real number when specifying an index) of the element required (counting along, left-to-right, or top-to-bottom, depending on the vector orientation), e.g.

```
» B(5)
ans =
    2.5000
```

or:

```
» C(3)
ans =
    1.5000
```

(In this text, I will refer to accessing a particular element (or elements) of a vector (or array) via its index as addressing. Unless I forget, then I might say something else. You'll have to keep on your toes – don't expect consistency here!)

There is a MATLAB function `end` (see Box) that enables you to easily address (accessing via its index) the very last value in a vector (in MATLAB, the index of the first position is always `1`).

For addressing more than one element of a vector at a time, you can use the `colon operator` (see Box).

As well as reading out an existing value of a vector, you can also replace an existing value by assigning the new value to the appropriate index position. e.g. to replace the first element with a value of `0.0`:

`length`
You can determine the length of a vector A with ...

```
length(A)
```

returning its integer length, and which could in turn be assigned to a variable, e.g. `B = length(A)`. (Technically, `length` returns the largest dimension of an array.)

`size` (use #1)
Returns both dimensions, even though for a vector, one of them always has a value of `1`. This does allow you to determine its orientation though, as for the example of `A = [1:10]`:

```
» size(A)
ans =
    1 10
```

(1 row and 10 columns). For `A = A'`:

```
» size(A)
ans =
    10 1
```

(10 rows and 1 column).

`flipud`, `fliplr`
These two functions allow you to re-order a vector. Their use is simple:

```
» B = flipud(A)
```

will invert the order of elements of a column vector, and:

```
» B = fliplr(A)
```

will invert the order of elements of a raw vector. Simples! Lesson over.

The `transpose operator`, in **MATLAB**-speak, "*returns the nonconjugate transpose of A*". Who knows what that means. In slightly more everyday (i.e. down here on Earth) language, it: "interchanges the row and column index for each element". Or sort of, just interchanges the rows and columns. The operation can be written:

```
» B = A.'
```

or

```
» B = transpose(A)
```

In practice, you can get away with being lazy (and in fact this is how it was in the old days, and just write):

```
» B = A'
```

(but get into the habit of using the formally correct, **Mathworks** official and UN-approved, syntax of `.'`).

```
B(1) = 0.0
```

(Here, you are saying that you would like to assign the value of `0.5` to the element in the vector given by the index `1`. The previous content of the array at index position `1` is simply over-written.)

You can access more than a single element of a vector at a time, by means of the `colon operator`, `:` to define a min, max range of indices. For example:

```
» B(2:4)
ans =
1.0000
1.5000
2.0000
```

To select all elements:

```
» B(:)
ans =
0.5000
1.0000
1.5000
2.0000
2.5000
```

`end`
Represents the largest index in a vector when addressing it, or in MATLAB-speak: "end can ... serve as the last index in an indexing expression".

1.4 *Basic graphing (aka. 'data visualization')*

So far ... I suspect this is heavy-going and there is a lot to try and remember, such as command names, although knowing just that certain commands exist, is enough to start with and MATLAB Help can be used later tot find out the exact name (and usage syntax). All this, and we have not even gotten on to matrices (2-D arrays) yet ... So, we'll take a diversion to look at some basic plotting techniques that will make sense now that you can create vectors of numbers to plot (and later, important some 'real' data). Unless you have forgotten how to create vectors already ... :(

1.4.1 *Plotting*

The command `figure` creates a figure window, which is where MATLAB displays its graphical output ... but on its own, without anything in it ... useless. So, lets put something in it, with the simplest possible graphical way of displaying data called `plot`. But first – create yourself a dummy dataset to plot. You are going to need to create yourself a pair of vectors – these can have any values (all numbers though) in them that you like, but perhaps aim for 1 vector with values counting up from 1 to 10 – this will form your *x*-axis, and the 2nd column ... whatever you like. [19]

As always, refer to the MATLAB `help` text on `plot` before using it (also refer to Box). The key information that will get you started is at the very top:

```
PLOT(X,Y) plots vector Y versus vector X.
```

So, you need to pass it your *x*-axis data vector (by its variable name), followed by your y-axis data vector (by its variable name) – comma separated. Do this, and depending on just what or how random your *y*-axis data was, you should end up with something like Figure 1.1 in a window captioned "`Figure 1`".[20]

This ... is easily the least professional plot ever. And one that breaks all the most basic rules of scientific presentation, such as an absence of any labelling axes. There is also no title, although here in the course text I have added a figure caption in the document so I can sort of get away with it. But this is the default state of the basic plot function and you'll just have to deal with it (i.e. add a series of commands to add missing elements of the plot).

Note that by default, MATLAB also scales both axes to reasonably closely match the range of values. In the example here, the default min and max axes limits in fact turn out to be the min and max values in the *x* and y-axis data because the data is composed of

[19] Looking ahead – you could create a y-axis vector formed of the squares of the numbers in the x-axis vector:

```
» Y = X.^2
```

(The `.^` bit says to square the value of each and every element in the vector.)

`plot`

The MATLAB function `plot` ... plots. More specifically, it plots pairs of (x,y) data and by default, does not plot the points explicitly but joins the(x,y) locations up by straight line segments. MATLAB calls these a '2D line plot', although there are plotting options that allow you only to display the individual (x,y) points (making it like the `scatter` function, which we'll see later).

Its most basic usage is:

```
plot(X,Y)
```

where `X` and `Y` are vectors – of the same length (important), but not necessarily of the same orientation (i.e. if one was a row vector and one a column vector, MATLAB would work it out, although it is perhaps best to avoid such a situation arising).

There are many options that go with this function, some of which we'll see and use later. You can also input matrixes as `X` and `Y` apparently. But I have absolutely no clue as to what might happen. I suspect that the plot will end up looking like a bad acid trip.

[20] If you cannot see the figure window ... check that the window is not hidden behind the main **MATLAB** program window!

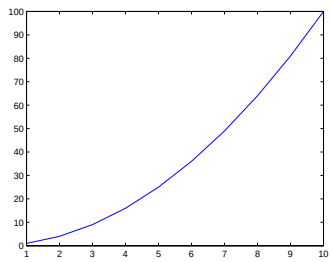

Figure 1.1: Default output of `plot`.

relatively simply/whole numbers. If however the maximum y value was vary slightly larger, you'd see that MATLAB would adjust the maximum y-axis limit to the next convenient value so as to preserve a relatively simple series of labelled tick marks in the axis scale. In fact, why not try that – replace your maximum data value, with a value that is very slightly larger (an example is given in Figure 1.2). [21] Then re-plot and note how it has changed (if at all – it will depend somewhat on what data you invented in the first place).

[21] If you have created a dummy dataset in which the value in the last row is the largest, replacing it is simple – remember the use of `end` in addressing an element in an array. If your dataset does not monotonically increase and the largest value falls somewhere in the middle ... you could cheat' and open the array in the variable editor and discover which row it occurs on.

1.4.2 *Graph labelling*

You have two options for editing the figure and e.g. adding axis labels. Firstly, you can use the GUI and the series of menu items and icons at the top of the Figure window to manipulate the figure. I suspect you'll prefer this ... but it is not very flexible, or rather, it requires your input each and every time you want to make changes or additions to a figure. The second possibility is to issue a series of commands at the command line. (The advantage with the latter we'll see later when we introduce `m-files`.) For now, I'll illustrate a few basic commands:

1. The first, obvious thing to do is to add axis labels. The commands are simple – `xlabel` and `ylabel`. They each take a string as an input, which is the text you would like to appear on the axis. If you change your mind, simply re-issue the command with the text you would like instead.
2. The command for title, perhaps unsurprisingly, is `title`. Again, pass the test you would like to appear as a string (in inverted commas "), or pass a the name of variable that contains a string (no '' then needed).
3. You might want to specify the axis limits. The command is `axis` and it takes a vector of 4 values as its input – in order: minimum x, maximum x, minimum y, and maximum y value. e.g. `axis([0 10 -100 100])` would specify an x-axis running from 0 to 10, and a y-axis from -100 to 100.

Information as to how to use all of these commands can be found via MATLAB `help`. But a typical sequence, that gives rise to the improved plot shown in Figure 1.3, is given in the margin.

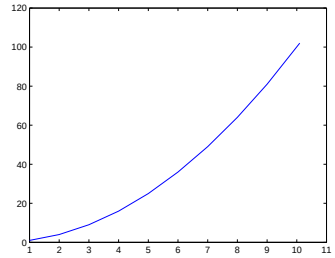

Figure 1.2: A plot illustrating axis auto-scaling (maximum x and y values now slightly larger than 10 and 100, respectively).

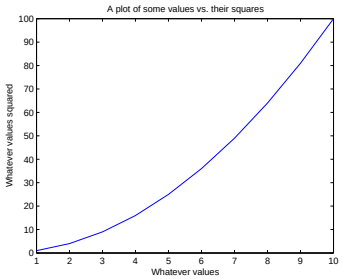

Figure 1.3: A (only very slightly) improved plot.

Example of adding axis labels and a plot title ...

```
» xlabel ...
  ('Whatever values');
» ylabel ...
  ('Whatever values
squared');
» title ...
  ('A plot of some ...
  values vs.  their ...
  squares');
```

1.4.3 *Sub-plots*

You can also have more than one plot in a single Figure window. As an example, create some sine waves using the `sin` function (see `help`) over the range $0 < x < 2\pi$, e.g.:

```
» x = 0:0.1:2*pi;
» y = sin(x);
» y2 = sin(2*x);
```

(Note how in the first line, the colon operator is used to create an x vector from 0 to 2π, in steps of 0.1. The second and third lines calculate the sine of all the x values, and sine of 2 times the x values, respectively, and assign the results to a pair of new vectors, y and y2.)

To place several different plots on the same figure uses the `subplot` command [22]. The `subplot` command is used as: `subplot(m,n,p)` where `m` is the number of rows of plots you want to have in your figure, `n` is the number of columns of plots in your figure, and `p` is the index of the plot you wish to create (see: Figure 1.4).

The basic code then goes something like:

```
» figure(1);
» subplot(2,2,1);
» plot(x,y);
» subplot(2,2,2);
» plot(x,y2);
» subplot(2,2,3);
» plot(x,-y);
» subplot(2,2,4);
» plot(x,-y2);
```

In this case, the 3rd and 4th subplots simply display the inverse of the curves in the subplots above.

[22] » help subplot

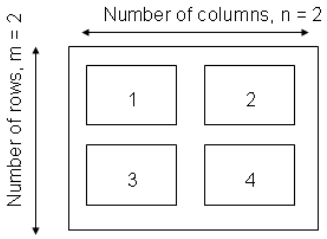

Figure 1.4: Arrangement of subplots.

1.4.4 *Saving graphics and figures*

You might just want to save the figure. (Why create it in the first place in fact if you are just going to throw it away ... ?) Again, you can do this via the GUI or at the command line [23]. From the GUI, you have the option to save the figure in a way that can be loaded later and re-edited – this is the `.fig` format option. Or you can save (export) in a variety of common graphics formats (although once saved in this format, the graphics can only be edited later using a graphics package).

You can also close figure windows (see Box). No seriously. They are not forever. ;)

[23] To export a graphic at the command line, use the `print` function. To cut a long story short (see: `help print`), to print to a postscript file:
`print('-dpsc2', FILENAME)`
where `FILENAME` is the filename as a string or a variable containing a string.

To close the current (active) Figure window, the command is:
`» close`
To close all currently open Figure windows:
`» close all`

1.5 Vectors and arrays #2

A matrix is another special case of an array – this time 2-D (rather than 1-D in the case of a vector). MATLAB totally hearts them.

1.5.1 Creating matrices and arrays

You can enter matrices (2-D arrays) into MATLAB in several different ways:

1. Enter an explicit list of elements. To enter the elements of a matrix, there are only a few basic conventions:
    - Separate the elements of a row with blanks or commas.
    - Use a semicolon, `;` , to indicate the end of each row.
    - Surround the entire list of elements with brackets, `[ ]`.
2. Load matrices from external data files.
3. Generate matrices using built-in functions.

As an example, type in the following at the command prompt:

```
A = [15 7 11 6; 13 1 6 10; 21 17 5 3; 5 15 20 9]
```

MATLAB then displays the matrix you just entered[24]:

[24] Remember that you can add an `;` to the end would prevent the assignment being displayed.

```
A =
15 7 11 6
13 1 6 10
21 17 5 3
5 15 20 9
```

Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as A.

Now go find the array you have just created in the *Workspace* window. Double-click on its name icon and see what goodies appear on the screen. This is a fancy array editor which looks a bit like one of those dreadful spreadsheet things. You can see that this might be handy to edit, view, and keep track of at least moderate quantities of data. This is a useful facility to have. However, we are going to concentrate on the command-line operation of MATLAB in the Lab because that will give you far more power and flexibility in applying numerical techniques to problem solving, and will form the basis of scripting (computer programming by another name) that we will see in a few lectures time. Close down this nice toy to leave just the original windows.

Elements in the matrix can be addressed using the syntax:

```
A(i,j)
```

where `i` is the row number, and `j` is the column number. It is very very easy to keep forgetting in which order the rows and columns are indexed., but I'll tell you here and now before I forget:

rows, columns

(You can always create a test matrix and access a specific element to check if in doubt!) In the example above:

```
» A(1,3)
ans =
    11
```

(i.e. the value of the element in the 1st row, 3rd column, is 11).

In general, the same functions and operators that applied to vectors and you saw earlier, also apply to matrixes (or specific dimensions of matrices). (See Boxes.)

Finally – a fundamental way of accessing data that you need to learn and be familiar with, is to employ the color operator to select specific columns (or rows) of data. You'll find that this skill ends up inherent to many of your attempts to process and graph data. For instance, if your (*x*,*y*) data to plot ended up in MATLAB workspace in matrix form (it very commonly does) rather than as 2 sperate vectors (as you had when you first plotted anything), you will need to select separately the *x* (e.g. 1st column) data, and the *y* (2nd column) data, and pass these to the `plot` function. For the example of matrix A above, all the first column data can be selected by typing `A(:,1)`[25], which says all the rows (`:`) in the first column. Similarly, all the 2nd column data alone can be selected by `A(:,2)`. (You'll practice this endlessly later on and hopefully get it!)

Similarly as for vectors, you can access more than a single element of a matrix by means of the `colon operator`, `:`. For example:
`A(:,1)` – selects the 1st column
`A(3,:)` – selects the 3rd row
`A(2:3,2:3)` – selects the 2×2 matrix of values lying in the centre of `A`, while `A(1:2,:)` selects the top half (first 2 rows) of the matrix.

[25] Remembering the HUGE hint above in 100 pt font as to the order of rows and columns ...

1.5.2 Basic matrix manipulation

You can treat vectors and matrices (or parts of vectors and matrices), mathematically, as you would treat single values (i.e. *scalars*) but unlike a scalar, the transformation is applied to all specified elements of the array. This applies for all the basic numerical operators[26]. For example, for vector `B` in the earlier example,

```
» 2*B
ans =
    0 2 3 4 5
```

and

```
» B-1.5
ans =
    -1.5000 -0.5000 0 0.5000 1.0000
```

You can also determine the shape of your array using the `size` function. For a 2D array (matrix), when you pass it the name of your array, it returns the number of rows followed by the number of columns (in that order).

[26] Technically ... or at least to be consistent with other operations, you might write multiplication as `.*` rather than just plain old `*`. The preceding dot tells MATLAB not to treat this as matrix multiplication but to carry out the operation on each element in turn. In this case, it is the same thing (and both notations work the same), but later, is not. (This will make more sense when you get to see it in action, later.)

QUESTION: Multiply all the elements of `A` by the number `17`. Assign the answer to a 3rd array (`C`). What is the value of the element `C(2,3)`? How would you ask for the 4th row, 2nd column element of the array `C`, and what is its value?

QUESTION: What is the sum of the 4th column of `C` ? (Sure – you also do it by using a calculator, but you will not always have such a small data-set as here. Perhaps you'll get a much larger data-set in the assessed exercise ;) So, practice doing it properly.)

QUESTION: What is the sum of the 2nd row of `C`? `sum` gives returns the sums of each column, and so on its own;

```
» C
C =
255 119 187 102
221 17 102 170
357 289 85 51
85 255 340 153
» sum(C)
ans =
918 680 714 476
```

gives you a row vector consisting of the sums of the individual columns of the matrix `C` above.

This is where the `transpose` function (`'`) comes in handy (see earlier). In this case, it flips a (2D) matrix around its leading diagonal (columns become rows, and rows, columns)[27] .

```
» C'
ans =
255 221 357 85
119 17 289 255
187 102 85 340
102 170 51 153
```

(transposing the matrix turns the rows into columns)

```
» sum(C')
ans =
663 510 782 833
```

Now you get a row vector consisting of the sums of the individual columns of the matrix C, but since you have transposed the matrix `C` first, these four values are actually equal to the row sums.

Finally, you could transpose the answer:

```
» sum(C')'
ans = 663
510
782
833
```

The *function* `sum` ... sums things. The MATLAB Help documentation (`help sum`) says:
`'If A is a vector, sum(A) returns the sum of the elements.'`
`'If A is a matrix, sum(A) treats the columns of A as vectors, returning a row vector of the sums of each column.'`

[27] This is almost true. Technically the function you want is `.'`, as `'` will change the sign of any imaginary components. For real numbers, they are the same.

In addition to `transpose`, other useful array manipulation functions include:
`flipup` – flips the matrix in the up/down direction
`fliplr` – flips the matrix in the left/right direction
`rotate` – rotates the matrix
(As always, refer to the `help` on specific functions.)

now with a row vector gives you a format that looks like the row sums of the original matrix `C`. [28]

1.5.3 Some matrix math :(

We will not concern ourselves with multiplying vectors and matrices together ... just yet ...

[28] Note how you can combine multiple functions in the same statement to create `sum(C')'`. However, to start with, it is much safer to do each step separately and hence be sure what you are doing.

1.6 Loading and saving data

There are a number of different ways to load/import data into the MATLAB *Workspace*. Rather than try and tediously list and describe the commands and syntax and blah blah, we'll be going through a couple of (hopefully) slightly-less tedious data-based examples as we progress through the course text. In this way, if nothing else, you might accidently learn some 'science' even if nothing much about MATLAB ...

1.6.1 Where am I?

Before anything – you need to know where you are. If your file to load is not in the directory MATLAB us using, it will not find it. And if you save something and have no idea where it is being saved ... that can hardly go well.

By default, MATLAB looks to a file directory located within its installation directory (`$MATLAB/data`). So, where the `load` command requires a filename to be passed, you will need to enter either the full location of the file; i.e., starting with the drive letter (e.g. as per displayed in the **Windows Filemanger** address bar, or the relative path to where the file is located (e.g. if there is a subdirectory called data, you will pass `data/sediment_core_d18O.txt`[29]). Alternatively, you can change the MATLAB directory that you are working in. (This works similar to UNIX/LINUX for those of you who are familiar with navigating your way around these operating systems.) You can make the download directory the default directory for working from by typing:

```
» cd DIRECTORY_PATH
```

where `DIRECTORY_PATH` is the path to the directory in which the data file has been saved[30], remembering that `DIRECTORY_PATH` is a string (i.e. enclosed in "). Or ... you can add a 'search path' (`addpath`) so that MATLAB knows where to look. (Note that both these alternative possibilities can be implemented from the GUI.)

There is also, of course, the GUI – from the `File` menu the option `Import Data...` will run the data import Wizard – note that you might have to select `All Files (*.*)` from the file type option box in order to find the file. I'll leave you to work the rest out for yourselves ... Maybe try importing the data into MATLAB this way once you have done it successfully using the load function at the command line.

[29] Remember that this is a *string* type.

[30] You can view the files that are present in the directory that you are working in by typing (more LINUX-speak): `ls`.

`load`
Loads variable from a file into the workspace. The syntax is:

```
» load(filename)
```

where filename is the name of the file (remember: as a string, it needs to be enclosed in quotation marks). The file might be plain text (ASCII) or a MATLAB workspace file (see below), in which case it should have the file extension `.mat`. To force MATLAB to treat the file input as ASCII or a MATLAB workspace file, pass a second parameter (separated from the filename by a comma) – `'-ascii'` for ascii, and `'-mat'` for a MATLAB workspace file.

Note that in loading an ASCII data file, any line starting with a `%` is ignored. Also note that the data must be in a column format with no missing data.

For an ASCII file, the name of the variable created to hold the data being imported is automatically generated. So in the example of the data file being called `'twilight.txt'`, the variable generated will be called `twilight`. You cna instead chose to assign the imported data to a variable name of your choice, by e.g.:

```
» sparkle =
load('twilight.txt');
```

1.6.2 *Loading and importing data*

The simplest way (other than via the MATLAB GUI and the beautiful green **Import Data** icon) is to use the `load` function (see Box)[31].

As a brief exercise and practice using `load` – first download the data file `etheridge_etal_1996.txt` from the course webpage of `www.seao2.org`. You might start by viewing the contents of the file by opening it in any text viewer. This is always a good place to start as it enables you to see what you are getting yourself in to (i.e. format of the file, any potential formatting issues, approximate size and complexity of the dataset, etc). Then import the data into the MATLAB workspace using the `load` command. Try simply typing the name of the variable that was automatically created (or the one you chose, if you assigned the imported data to a specific variable name – see Box) to provide a crude view of the data. Then double click on the name of the variable in the MATLAB Workspace window. This should open up a spreadsheet-like window in which the data can be viewed, sorted, and even edited. Finally, plot the data and remember to label it appropriately[32]. You should end up with something like Figure 1.5.

[31] There is also a much more flexible way of loading text-based data using the function `textscan`, but that also requires files to be explicitly opened and closed using `fprintf`. We'll see a little of this later.)

[32] FYI: the x-axis data is year, and the y-axis data is the mixing ratio of CO_2 in air in units of ppm.

1.6.3 *Saving and exporting data*

Arrays of numbers can be saved in a plain text (ASCII format) by means of the `save` function in a simple reverse of the use of `load` (see Box).

1.6.4 *Loading and saving the workspace*

The entire workspace (including all variables and their values, or just the values in a single variable if you wish) can be saved to a file and then later re-opened. The file format is specific to the MATLAB program and the file-name extension by default is `.mat`. You might find this very helpful to use in long lab exercise or large modelling projects, particularly if you do not come back to work at the exact same computer each time or wish to use continue the same piece of work on a laptop elsewhere.

`save`

Saves variables from the workspace to a file. There are two main forms (syntaxes) of the command:

```
» save(filename)
```

which saves the entire workspace to a `.mat` file (with the filename given by the string `filename` (in quotation marks), and:

```
» ...
  save(filename,A,'-ascii')
```

saves the data in the variable `A` (which must be given as a string, i.e. also enclosed in quotation marks) in plain text (ASCII) format.

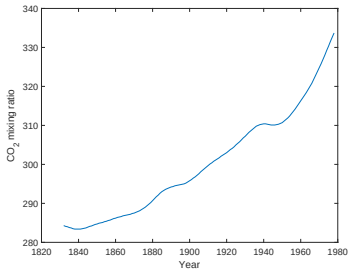

Figure 1.5: Spline fit to measured changes in CO_2 concentration in Law Done ice core, following *Etheridge et al.* [1996].

1.7 *Basic data processing*

As an example to kick-off some data-processing tricks, load in the Phanerozoic CO_2 dataset: `paleo_CO2_data.txt`. You can just import it into MATLAB using the `load` function. However, there is a complication here – unlike the ice core CO2 dataset, you now have 4 columns in the array[33]. The first column is age (Ma), the second the mean CO_2 value, and the 3d and 4th are the low and high, respectively, uncertainty limits. Remembering (I hope!) how to reference specific columns of data in a matrix[34] – plot the mean paleo CO_2 value as a function of age (in Ma). If you closed the previous Figure window (see earlier), it is not essential to explicitly open one (using the `Figure` command) – when you use the `plot` command, if there is no open Figure window, **MATLAB** will kindly open one for you. How thoughtful. The result should be something like 1.6. O dear ...

So ... that was not so successful. What is happening in the default behaviour of `plot`, is that the location defined by each subsequent row of data is being joined to the previous one with a line. This was fine for the ice-core CO2 example dataset because time progressed monotonically in the first column, e.g. the data was ordered as a function of time. If you view the paleo CO_2 data, this is not the case. (In fact, in the original, full version of the data, ordering is by proxy type first, and then study citation, and only then age ...).

Your options are then:

1. You could import the data into **Excel**, then re-order (sort) it, then export it, then re-load it ...

2. You could sort it in **MATLAB** using the GUI variable view window. But lets not cheat for now.

3. You could sort it in **MATLAB** at the command line. How? Well, a reasonable gamble, which actually turns out to be a total win, is to try:

```
» help sort
```

Actually ... not quite. Reading the help text carefully (and you can always try it out and see what exactly it does if you are not sure), `sort` will sort all columns independently of each other, whereas we want the first column sorted and the remaining columns linked to this order. Under `see also`, MATLAB lists `sortrows` as a possibilty. The help text on this looks a little more promising. It is still slightly opaque, so the best thing to do is to try it (and view the results)! This looks rather better. The resulting of `plot`-ting this is 1.7. (This is a good illustration of a guess of a function that was

[33] Remember that you can diagnose its size with ... `size` (or refer to the Workspace window)

[34] HINT: the `colon operator` (see earlier).

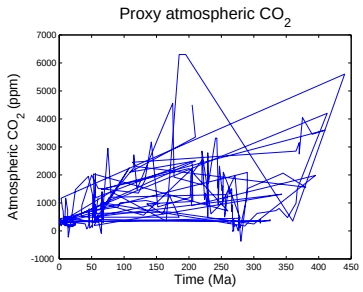

Figure 1.6: proxy reconstructed past variability in atmospheric CO2.

not quite what was needed, but following up on the `help` suggestions leads to a more appropriate function.) At least now the curve is reminiscent of past changes in global temperature and the geological Wilson cycle, with high values in the Cretaceous and Jurassic and then lower again in the Carboniferous (roughly matching the progression of ice and hot house (and then back to recent ice ages) climates).

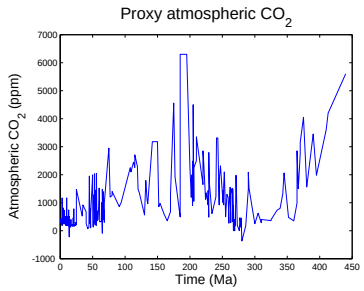

Figure 1.7: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

As a second example and to get you familiar with some additional very basic data processing, we are going to transform a sediment core $\delta^{18}O$ time-series into an estimated history of glacial-interglacial changes in sea-level. The scientific backstory is ...

Throughout the late Neogene[35], sea level has risen and fallen as continental ice sheets have waned and waxed. The main cause of sea-level change has been variation in the total volume of continental ice and resulting change in the fraction of the Earth surface H_2O contained in the ocean. Today more than 97% of the Earth surface H_2O is in the ocean and less than 2% is stored as ice in continental glaciers, with groundwater making up the bulk of the remainder. Of the total continental ice (ice above sea level), 80% is contained in the east Antarctic ice sheet, 10% in the west Antarctic ice sheet, and the final 10% in the Greenland ice sheet. (If all present-day continental ice were to melt, sea level would rise by 70 m.) During the last glacial maximum (LGM), sea level was about 125 m lower than present, equivalent to 3% more surface H_2O stored as continental ice. Because of its relationship to continental ice volume, an accurate late Neogene sea-level curve has been a long-term goal of scientists interested in ice-age cycles and their causes.

[35] 23.03 millions years ago (end of the Oligocene) to present is the Neogene Period in Earth history.

Glacial ice has a lower $^{18}O/^{16}O$ isotopic ratio than mean seawater[36]. When ice volume is high, seawater has relatively high $^{18}O/^{16}O$ ratio. When ice volume is low, seawater has relatively low $^{18}O/^{16}O$ ratio. If the average $^{18}O/^{16}O$ ratio of glacial ice is constant with time, then changes in the average $^{18}O/^{16}O$ ratio of seawater will linearly approximate changes in the total volume of ice and by inference, sea-level. We (at least, I am) are interested in all this because knowing how ice volume and sea-level changed over the glacial-interglacial cycles has all sorts of important implications for understanding how climate change (e.g. via ice sheet albedo) and global carbon cycling and atmospheric CO2 (e.g. via changes in the area of exposed continental shelves and carbon stored in soils and above-ground vegetation).

[36] Basically – as moisture derived from the tropical ocean (and land) surface moves to high latitudes, condensation occurs and some of the moisture is lost as rain. In condensating water vapor, ^{18}O is preferentially incorporated into the liquid phase, meaning that the remaining water vapour has lower $^{18}O/^{16}O$. Eventually, the residual water vapour might fall as snow on an ice sheet. Hence why ice sheets at the LGM will have a lower $^{18}O/^{16}O$ than mean seawater.

To start with we need to reconstruct past changes in the oxygen isotopic composition of the ocean. Handily, the $^{18}O/^{16}O$ ratio of foraminiferal calcite isolated from marine sediments is primarily a function of the $^{18}O/^{16}O$ ratio of the water together with the tempera-

ture of the water[37]. By measuring the $^{18}O/^{16}O$ value of calcite downcore we are sampling $^{18}O/^{16}O$ with a progressively older age. In this way we can reconstruct how ocean $^{18}O/^{16}O$ has changed over time. These measurements are reported in units of parts per thousand (‰) and written as $\delta^{18}O$.

How to turn (scale) changes in $\delta^{18}O$ into sea-level change? Evidence from dated coral reef terraces suggest that sea-level was around 117 m lower at the peak of the last glacial (ca. 19 ka). We could then assume that the change in $\delta^{18}O$ from modern (preindustrial) to LGM equates to 117 m sea-level change, and hence create a continuous past sea-level curve from all the $\delta^{18}O$ data by applying a simple scaling factor[38]. So:

- You first need the foraminiferal calcite $\delta^{18}O$ data. (Unless you want to go drill a long cylinder of mud from 3000 m down in the Atlantic Ocean, pick out all the microscopic foraminifera of a single species from samples of mud that you have carefully washed, blah blah blah ...) So, from the course web page; download the file `sediment_core_d18O.txt` and save it locally.

- Load this file into MATLAB.

- If you have successfully loaded in the data-file, you should see a named icon for the array appear in the `Workspace` window. Try viewing the file in the two different ways:

  1. At the Command line (»), type in the array name.
  Because of the length of the data-file we imported, the contents of the array should have whizzed past you on the screen in a highly inconvenient fashion. You can use the scroll bar on the right of the Command Window window to move up and view the data that you can't see (the younger age $\delta^{18}O$ numbers). Note that as MATLAB imports data into an array from a file, it names the array it creates following that of the filename, but without the extension (the '.txt' bit).

  2. Double-click on the array's icon in the Workspace window. Marvel at the fancy spreadsheet-like things that appear. Note that you can edit the data, add and delete rows and columns, and all sorts of stuff in this window, just like you can in Excel. Amuse yourself by scrolling down to the end of the data-set in the Array Editor and adding a new piece of data on line 784; age (column 1); 783 (ka); sea-level (column 2); 0.0 (m).

At the command line, list the contents of the array again to view the change you have at the end of the data-set. Use the **up arrow** to bring up the command you want rather than typing it in again.

[37] We we will not concern ourselves with temperature corrections here (in any case, it turns out that the temperature effect has the same sign as and is closely related to the ice volume effect) but instead assume that foraminiferal calcite $\delta^{18}O$ only reflect changes in (global) ice volume and sea-level.

[38] Conceptually, this is no different from saying that the difference between the freezing and boiling point of pure water (at 1 atm pressure) on the Celsius scale – 100°C, maps onto the equivalent interval on the Fahrenheit scale – 180°F (212-32 °F), and hence providing a means of converting a record of past changes in Fahrenheit, inot degrees Celsius (and *vice versa*).

Now delete this new row. Note that it is easy to get confused with which row number you need to address – although the data starts from year 0, MATLAB always counts the index (the sequential integer counting of the row or column number) of a location from 1 (one). (So age 10 ka is on line 11, and age 200 on line 201, etc.)

Reminder: for a $n \times m$ array `data`, the first row is:
`data(1,:)`.
The last row is:
`data(end,:)`.
To find out the number of rows is:
`» length(data)`.
The total size, in rows×columns, can be found by:
`» size(data)`
(and also by referring to the **Value** column in the **Workspace** window)

- So far everything has been in δ^{18}O units and time as kyr. As a warm-up – try converting the units of time to years by multiplying the first column of the data array by 1000.0 and assigning it back into itself (this is not as weird and nonsensical as it sounds).

To estimate past changes in sea-level we need to scale the δ^{18}O values to reflect the equivalent changes in sea-level rather than changes in isotopic composition. We know that sea-level is 0 m (relative to modern) at 0 years ago and -117 m at 19,000 years ago. Try the following (you are going to have to *think*, but maybe also use the HINT in the margin):

Scale the δ^{18}O so that it represents changes in sealevel, relative to modern (0 m)[39].

- Plot it (changes in sea-level compared to modern, vs. time).

And **nicely**.

[39] HINT – first determine the difference in δ^{18}O between time zero and 19 ka. This gives you the range of δ^{18}O that maps onto a sea-level change of 117 m. You also might transform the δ^{18}O data such that it has a value of zero at 0 ka (but retains the original amplitude of variability.

1.8 Yet more graphing

This section covers how to create slightly fancier plots in MATLAB and combines this with some more data loading practice.

1.8.1 Modifying lines/symbols in `plot`

The first deviant activity you can engage in with `plot`, it to graph the data without the line joining the points. Scrolling a little the way down » `help plot`, it turns out that there are a number of options for color, line style, and marker symbol that you list together as a single parameter, straight after the parameters for x and y vectors. By default, MATLAB plots a solid line in blue with no marker points. Obviously, we could forego the sorting and plot a sane graphic (hopefully) by plotting just points and having no line between them. Hell, you could even be radical and use a different color ... Or, you could specify a symbol and no line. The choice of colors is your oyster, as they (almost don't) say. e.g. Figure 1.8.

The main (i.e. not an exhaustive list) data display options for the `plot` function are:
(1) point style
`.` – point, `o` – circle, `x` – x-mark, `+` – plus, `*` – star, `s` – square, `d` – diamond, `v` – triangle (down).
(2) line style
`-` – solid, `:` – dotted, `-` – dashed, and when specifying a point style, not specifying a line style results in no line.
(3) color
`b` – blue, `g` – green, `r` – red, `y` – yellow, `k` – black, `w` – white.

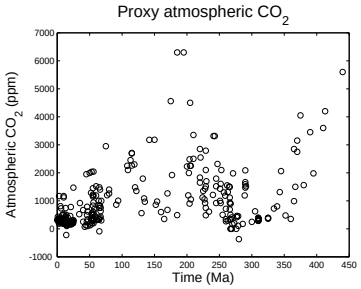

Figure 1.8: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

1.8.2 Plotting multiple data-sets

So far, so good. But so boring, although simple marker-only and joined-by-line plots have their place. For a start, the original data-set included an estimate of the uncertainty in the CO_2 reconstructions in the form of the min and max plausible value for each 'central' (best guess?) estimate. **Excel** can make plots incorporating errors, including non-symmetric errors, relatively easily. What about in **MATLAB**? Actually, I have absolutely no idea. (This would make such a good exercise for the reader, as they (do) say.)

Personally, I might have been tempted to draw vertical bars alongside the data (most likely). Or plotted in different symbols, the min and max values as points. Or plotted min and max lines as a bounding envelope. All of these require sone further little trick in MATLAB, which involves the command `hold`. This is nice and simple and can be `on`, or `off`.

» `hold on` – will enable you to add additional elements to a graphic,

» `hold off` – returns to the default in which a new graphic replaces the current on in a Figure window.

As an example – set » `hold on`, and then plot the minimum and maximum CO_2 values (columns #3 and #4) in different symbols and different colors, on top of your existing plot. If you want to then label what different lines or sets of points are, you can add a legend with the `legend` command. For instance you have managed to successfully

plot the mean CO_2 values as discrete black circles, and the minimum and maximum uncertainty limits as blue and red lines, respectively, you could call:

```
» legend('Mean CO_2','Lower uncertainty limit','Upper uncertainty
limit');
```

and it should end up looking like Figure 1.9.

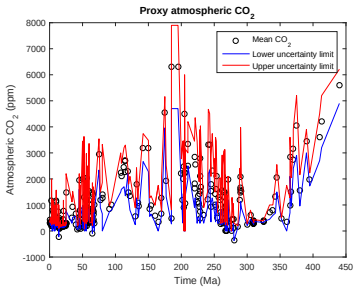

Figure 1.9: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

1.8.3 *Scatter plots*

We'll stay with the Phanerozoic proxy (CO_2) data, but put a different (graphical) spin on it.

Consider ... `scatter`. In fact, don't just considered it, `help` on it. The simplest possible usage is, apparently:

```
SCATTER(X,Y) draws the markers in the default size and color.
```

(where `X` and `Y` are vectors). This almost could not be more straightforward. Make yourself an `X` and `Y` vector out of the loaded-in dataset (or if you are feeling brave, you can pass in directly the appropriate parts of the dataset array), close the existing Figure window[40], and `scatter`-plot the (mean) CO_2 data.

[40] See earlier.

Perhaps a little disappointingly, the default (Figure 1.10) (plus added labels) looks a little like one of the plots before. However, `scatter` can plot color-filled symbols, but more powerfully, can scale the fill color to a 3rd data value (vector), in a sort of pseudo 3D *x*-*y*-*z* plot. For instance, it will be duplicating information that is already presented (*y*-axis), but you could color-code the points, by the y-value, i.e. the atmospheric CO_2 value. e.g.

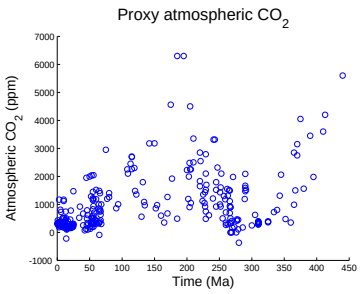

Figure 1.10: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

```
SCATTER(data(:,1),data(:,2),20,data(:,2))
```

draws the markers with an (area) size of 20 (points), in different colors. Coloring just the outlines of the circles is perhaps not ideal (difficult to see all of the color differences), so the circles can be filled in instead (and you could make them a little larger too):

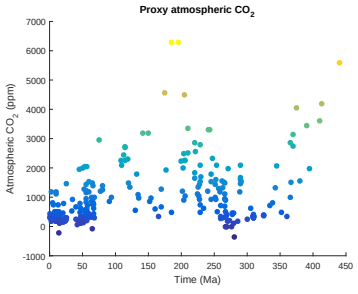

Figure 1.11: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

```
SCATTER(data(:,1),data(:,2),40,data(:,2),'filled')
```

resulting in Figure 1.11.

One final example in this section to introduce some new plotting functions, but also to quickly go back over some basic array manipulation and processing. The data we will be analysing is s series of seismic readings from the USGS. The quake data are extracted between -5 and 20 lat, and between 90 and 105 lon, starting Dec 26, 2004 and ending June 30, 2005. The data file can be found on the

course webpage (`data_USGS.txt`). The columns are: (1) day, (2) latitude, (3) longitude, (4) depth, and (5) magnitude. Carry out the following:

1. The first earthquake in the list is the Sumatra earthquake of December 26, 2004. The magnitude of this earthquake has been revised upward since the data was downloaded. Actually, most energy released in large earthquakes is in very low frequency shaking that most seismometers do not record. The real magnitude had to come from a special analysis of "normal modes", or standing waves on the Earth's surface with periods of up to 54 minutes! When the media said that the Sumatra earthquake made the Earth ring like a gong, these are the waves they were talking about. So since we know that the magnitude was really 9.3, first off, replace the value of the magnitude of the first earthquake in the array.

2. Identify the smallest magnitude of recorded earthquake. You should find that the minimum earthquake size on this list is 3.5. For an earthquake in California, the minimum magnitude would be more like 1. This is because this particular seismograph network did not have many instruments around Sumatra. Another problem is that the earthquakes are offshore. If the nearest seismograph is far from a small earthquake, that earthquakes may not be detected. This means that the data are artificially truncated. Since everything below 3.5 is missing, some of the M=3.5 to 4 earthquakes may have been missed, too.

3. Identify the minimum and maximum earthquake depths. The really deep ones (>40 km) are probably in the subducted slab that goes beneath Sumatra. The zero depth means that it could not be resolved - most hypocentres are 4 km or deeper. (hypocentre = like epicentre, but at depth: the point on the fault where the earthquake rupture starts)

4. How many earth quakes in total were recorded?[41]

There is only just so much looking at and processing raw data you can do before your eyes start ... to ... ... droop ... ... and ... ... Zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz. OK – so now to visualize what is going on. Plot using the scatter function the locations of all the quakes from day 0 to day 91 (inclusive), and in a second plot the locations from day 92 onwards. The first area covers the area that ruptured in the M 9.3 quake (1200 km long and 100 km wide) and the second, to the South, is smaller. This is important because the aftershock distribution made people very wary of the (low) early

[41] Recall how to find the size of an array. The number of earthquakes is then simply the number of rows (assuming that you have not flipped the array around ...).

The number of quakes bigger than each magnitude should go up by about a factor of 10 for unit decrease in magnitude (Gutenberg-Richter relationship, a power law). This fails for the hugest quakes (>7 in this case) and where the catalogue is incomplete (not many between 3 and 4 due to detection threshold in this part of the world).

magnitude estimates - the area of dense aftershocks often delineates the part of the fault that ruptured, and scaling laws relate rupture length to magnitude.

Create a figure with multiple panels, showing:

- In the top LH corner plot the day 0-91 quakes, and color-code (or size-code) the markers for their magnitude.
- In the top RH plot the day 92 onwards quakes, and color-code (or size-code) the markers for their magnitude.
- In the bottom LH corner plot day 0-91 quakes, and color-code (or size-code) the markers for their depth.
- In the top RH plot the day 92 onwards quakes, and color-code (or size-code) the markers for their depth.

1.8.4 Histograms

We could also visually analyse the data as a histogram. Type `help hist` in the Command Window for a description of the `hist` function. The histogram must be supplied with a vector defining the 'bins' in which to sum the data. Here is your chance to use the colon operator again. O happy day.

1. To plot the frequency distribution of quakes as a function of their magnitude we need to create a series of bins to define the different magnitude ranges. How about bins with boundaries at magnitude; 1.0, 2.0, 3.0, ... 10.0. One complication is that the values in the vector M define the middle of the bins in the hist function and not the boundaries. The mid-points of this will be; 1.5, 2.5, 3.5, ... 9.5, and this is the vector you need to create and assign to a vector `M` (i.e., a vector array starting at 1.5, ending at 9.5, and with increments of 1.0).

2. Having created `M`, plot the histogram of quake frequency vs. quake magnitude by issuing:

```
» hist(data_USGS(:,5),M);
```

Question: what is the most frequent magnitude range of 'quake?

3. Now plot the histogram of quake frequency against time (i.e., day number) up to day number 186. You will have to assign a new vector of values to `M`, one that starts at 0.5 and ends at 185.5. Omori's Law says that the number of aftershocks per day should decrease following a power law – does this look to be the case (approximately)? (One problem is that the small earthquakes are missing which makes it appear not to work so well!)

4. Try this again (i.e., frequency of quakes vs. time), but investigate the effect of changing the bin size – try making the bins about 1 month (30 days) in duration. Note that now M must start at 15.0 (the mid-point of the first monthly bin). Sometimes changing the bin size can help if the data is noisy, but sometimes you lose important information. Which was better do you think – can you still see a power-law decay in quake frequency following each major event with the data in monthly bins? If you want, experiment with other bin sizes to see how the data comes out. There is not always a 'right' answer in plotting data and sometimes you just have to experiment a little to see what looks good.

Don't forget that all the plots you make should be appropriately labelled ... Save them as a `fig` file if you think you might want to edit them again, and/or export as an image.

1.8.5 *Simple 2D data and bitmap visualization*

There are 2 different simple MATLAB commands for visualizing a 2D dataset (i.e. a matrix) as a bitmap image (and via a 3rd command, viewing various bitmap photo and image format files too). As something (2D data) to play with – load in the matrix `model_grid.txt`.

First off – as before, view the data in the array viewer, just to get a feel for what you are dealing with here (although you are unlikely to be much wiser after doing so). So go ahead and employ the `pcolor` function in its simplest possible usage (see Box). You can see (Figure 1.12) that it is ... something. Maybe a little like the continents, but up-side-down at the very least. What to do?

Well, it is a good job that you remember how to re-orientate arrays, right?[42] If you guess right first time (three different basic transformations of a matrix were described), you get Figure 1.13.

Next try something very similar. but using the `image` function. Now the model grid is the correct way around! I have absolutely no idea why and why it is reading the matrix dimensions differently from `pcolor`. I am sure you could Google and find out. But you would have to actually care first.

What is the point of this? So you have the ability to simply visualise a gridded dataset. Later, we'll be doing it properly and it gets rather more involved when you have to create matrixes to describe the grid dimensions (e.g. lon and lat) for yourself.

As your very last exercise – find an image on the internet that amuses you, download it, load it into MATLAB (using `imread`), visualize it using `image`, and then ... well, that depends on how amusing it is. Maybe try plotting something on top of it (using `hold on`) or simply go home.

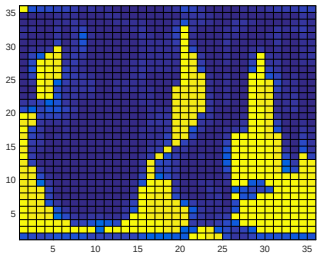

Figure 1.12: A 2D plot of some random gridded model data.

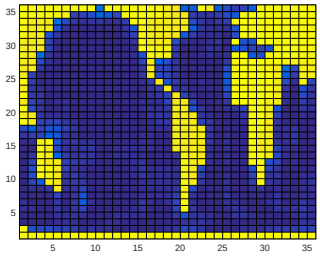

Figure 1.13: A 2D plot of some random gridded model data ... but with the underlying data matrix re-orientated before plotting.

[42] You don't? See earlier in the Chapter ...

pcolor

MATLAB claims that pcolor(C) plots; "a rectangular array of cells with colors determined by C. Actually, I believe MATLAB on this. So if you have a matrix, MATLAB will plot a regular arrays of cells, with each cell representing one of the elements in the matrix, and will color that cell according to the value. (`pcolor` will by default, autoscale how the color scale maps onto the data in the matrix such that both extreme ends of the color scale are used.)

image

You can import an image, such as in `.jpg`, `.tiff`, or `.png` format, using `imread` – simply pass it the name of an image file (as a string, this variable name needs to be encased in inverted commas) and assign the results to a variable name of your choice. Then plot (using `image`) that variable.

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 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 need to use the function `disp` (see Box or type » `help disp`) as always, for function syntax and usage), which will print to the screen, either any text you specify (in inverted commas), or the value of a variable (which could also contain character information). For now, simply pass the text directly. Your program needs just a single line in the **m-file**:

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

Save the file (to your working directory). Run it at the command line by typing its name (omitting the **.m** extension). Your first program

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

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

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

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

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

is a success! (Surely you could not screw up a single line program ... ?[5]) You could extend this to a mighty 2-line program by defining the string as a variable and displaying the contents of the variable, i.e.,

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

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

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

```
disp(X)
```

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

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

```
disp(X)
```

is ... well, find out for yourself!

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*).
- To start with – program in as a simple step-by-step way as possible. Breaking a complex calculation into several lines of simpler calculations is much easier to debug and work out what you were doing later, particularly if comments are also added. For all practical purposes – at this level, everything will run just as fast whether as a complex calculation on one line, or simple bite-sized calculation spread over 4 lines with comment sin between.
- Always save your changes before running your program (or you may unknowingly be running the previous version).
- If using the script to do some plotting, sometimes (but not always) it is convenient to add at the top of the `m-file`,

```
close all;
```

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

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.

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

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

which also illustrates one possibility for variable naming convention ('constants' (variables which never change in value) start with a `const_` and parameters (variables whose values might be changed) with `par_`, temporary ('local') variables with `loc_` and variables passed into and out of the function: `dum_`). Note use of the semi-colon at the end of every line to prevent (here unwanted) printing of results to the screen. In the file, you can create as much 'ASCII art' as you like if it helps to make the code clearer, e.g. adding separator comment lines ...

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

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

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

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

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

```
% program to print 'Hello World' to the screen
% *** START ***
```

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

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

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

2.1.2 *Debugging the bugs in buggy code*

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

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

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

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

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

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

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

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

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

Actually ... if this were your program, you should have paid attention to earlier and not have written it all at once before testing it! But

at least **MATLAB** is giving you some sort of feedback. The actual error reported might not always mean that much to you but the line number at which the problem occurred is gold-dust. The line of code is does not like is line 6[11], which is:

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

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

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

```
» cos[0.0]
```

to which you get told:

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

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

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

Other strategies for helping debug include:

1. Checking the what the values of the variables were at the point at which the program derped – the current (and the point of program crash) variable values are listed in the **Workspace window**.
2. Changing the relevant variable value(s) (here `x`) and re-typing the problem line to see if it makes a difference[12].
3. Commenting out (%) lines of code temporarily, or adding in additional (temporary) lines of code, and re-running. Where coding in bite-sized chunks is an advantage in this respect, is that if a program stops working after you have added a new section o code, you can go comment out the new code (never normally just delete it all), check that the original section of code still works, and then line-by-line, un-comment the new code until the problem line is found.
4. You can also put your program on hold just before the problem line and explore the state of the variables at that point (see Box),

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

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.

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

```
Undefined function or variable 'k'.

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

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

Once you have fixed the second bug; saved, and re-run the script, you should see Figure 2.1.

Debugging – breakpoints

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

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

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

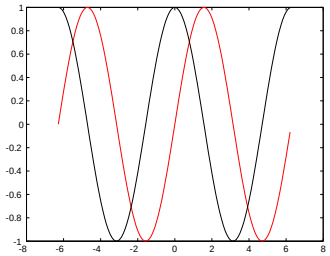

Figure 2.1: Output from the (bug-fixed version of) `plot_some_dull_stuff` **m-file**.

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 an output (in contrast, a script takes no input and returns no outputs, other than plots or data files that might be saved).

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

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

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

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

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

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

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

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

```
» sin2(0.5)
```

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

```
» Y = sin2(0.5);
```

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

Go create your own function now. Start by creating one that takes a single input and returns a value equal to the sine of the square of the value (rather than the square of the sine as above). When you are happy with this, create one with 2 inputs (see Box), that returns a

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

Functions

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

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

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

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

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

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

```
» calc_stuff()
```

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

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

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

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

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

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

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

```
end
```

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

value equal to the sine of the first input, divided by the cosine of the second input[15] (i.e. $y = \frac{sin(x_1)}{cos(x_2)}$).

You have used other functions, perhaps without knowing it, and some of them return values, but because you have not attempted to assume the returned values to anything, you have not noticed. For example, plot and scatter are in fact a functions, and return the 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, a variable `tmp` is created to hold the value of the partial calculation. It does not appear in the **Workspace window** when you use the function. The advantage of this is that you could create a second function that also created a temporary variable internally called `tmp` with both instances of `tmp` treated entirely sperate and isolated by **MATLAB** (i.e. setting the value of one instance of `tmp` does not affect the value of the other). This also however does lead to some additional complications in debugging *functions* (see Box). Try setting a breakpoint at the start of the line where the square of `tmp` is calculated – note that `tmp` now appear in the **Workspace window**. Continue the function and when it terminates, note that `tmp` is now gone from the list.

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

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 other main programming constructs is the conditional statement, in which the outcome (one or more statement(s)) is conditional on the 'truth' or otherwise of a given (i.e. it being true or false). 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.

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.)

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[16]. 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).

With this (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 to remind you what this script is going to do.

[16] 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

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`. 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 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`).

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 `answer` in the variable list and can 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.[17]

OK – aside from the use of `input`, there is nothing new here. Yet. The 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 will execute depending on the value of a variable – here 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 `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 '`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');
```

`input`
There are two variants – one for inputting numerical information and one for inputting a string (test) (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 `x` when a number is entered and RETURN 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 **RETURN** pressed. Note that the second parameter passed to the function `input` (`'s'`), tells **MATLAB** that the input is a string rather than a number.

[17] 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? ';`.

`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'.)

```
if comparison_result
  [MESSAGE]
end
```

Or, we could have made this more compact:

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

Your code should now have the 3 lines from before (comment, define question, get input) followed by 4 lines of the conditional structure, comprising: the `strcmp` function, the `if ...`, use of `disp` to display a message, and last, `end`.

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 `elseif` which would be followed by a line displaying the alternative message (e.g. 'Then you need to get a life, apple-lover.')[18].

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

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 2]
else
  [MESSAGE 3]
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.

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

```
if ~strcmp(my_answer, 'no')
  [MESSAGE 1]
else
  [MESSAGE 3]
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'.

Continuing to beat this same tired example to death ... what if some wise-crack answered 'YES' rather than 'yes'?[19] 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 ot 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[20] 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.[21] (You could extend it to 'no' also but I think you get the point ...)

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

[19] 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 ...

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

[21] 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'.

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!'.[22]

[22] 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 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 equaol to 10 (or equivalently, greater than 9), i.e.

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

or

```
if (my_answer > 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
      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
```

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
```

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[23]), 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 – loops cycle through a series of numbers between specific limits, or if you like, 'count'. 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'[24] (i.e. the code fragment) in a loop, later.

Consider, or rather: create a new **m-file**[25] with 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 without the loop to do anything.[26]

There are 2 alternative crude debugging strategies you could take[27]:

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

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

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

2. Print the value of `n` 'properly'[28], e.g.

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`.

[23] 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.

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

[25] Comment it!

[26] 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.

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

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

```
disp(n)
```

```
for n=1:10
   disp(str2num(n))
end
```

or you can tart this up even nicer by creating a string that provides more explicit information back to you, e.g.

```
for n=1:10
   my_string = ['The value of n is:  '  str2num(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:  '  str2num(n)])
end
```

(but they work the same – check it).

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).[29] The basic code is simple. Using the other (numerical input) form of `input`, for 2 numbers, it might look like (although in practice, your code is full of helpful comments, right?):

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

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

The first 4 lines you should be A-OK with. Note that in line 5, 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!

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?

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 = VAL1:VAL2
   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 (`VAL1`) to 4 (`VAL2`) 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. 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 (assuming that one of them was not also your solution). Note that you are note 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.

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`.

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`).

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 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.[30] 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:

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

```
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 and display the sum.

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. Equally, not only does the count not need to start at one, and the lower loop count limit could also be a variable (`min_count`?).

Finally, in addition to flexible loop count limits, the value of the increment in the count each time around the loop need not be one. 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)[31]. 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(?)

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

Create a new `m-file`. Call it ... anything you like[32]. 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. Also, for now, it is helpful to include the command: `close all` (which closes all currently open figure windows) although this is far from essential.

[32] `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`[33]. We should start by loading this little lot in. For the first file we could write:

[33] 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 data in it. Look at the **Workspace window** (or type `size(temp)`) – you now have a 3D ($94\times192\times2$) 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.

We first need to construct the loop framework. We'll call the month number counter variable, `month`. Create a loop (with nothing in it yet) with `month` going from `1` to `12`.[34] 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[35]. What happens? Can you tell?

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[36]). Within the loop, add the following line:

```
disp(month)
```

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

[34] 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 %.

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

[36] 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.

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[37] (unless you have grown rather attached to it) and move on.

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 for the first dataset 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).

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 some debugging[38]:

```
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.

comment out the `disp(filename)` line, and add a new line to load in each dataset from the new filename that is constructed each time the loop goes around.[39] Assign the new 2D data array to the `temp` array at the next layer number. Take a look at the **Workspace window** – note that you have an array (`temp`) that has size $94\times192\times12$. If temp is $94\times192\times1$ 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.)[40]

[37] 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.

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

[39] 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.)

[40] 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 2.2[41]. (Don't just type this line in blindly (maybe it doesn't 'work' anyway). 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?[42] Note that this is where the close all command at the start of your script comes in useful. Because if you re-run the script, you wont then end up with 24 figure windows. And then 36 the time after that, and ... (There is actually 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[43]:

```
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[44].

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.

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

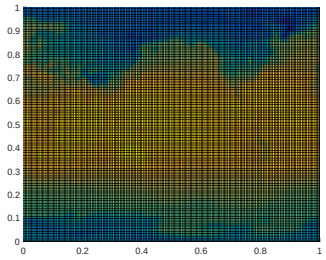

Figure 2.2: Extremely unappealing blocky plot of Earth surface temperature (who cares with month? – the graphics are too poor to matter ...).

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

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.

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

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

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[45].

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[46]. 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).

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

[46] **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 in saving a data file, to test for a filename already existing and if so, automatically modifying the new file name so as not to over-write the file.[47] 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[48]. 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[49] 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.

[47] 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.

[48] 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.

[49] 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[50], and assign the new (longer) string back into the variable `my_filename`.

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

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).

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[51]:

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

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

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')[52]. 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

[52] 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 ....[53] 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.[54]

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 check against the directory contents. What is missing is the initial value of `filename_check`. In a previuos example, we simpy 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 veyr 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
```

[53] 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.

[54] 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`) 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 2.3.

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 2.4..

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 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
continental_outline_end.dat
```

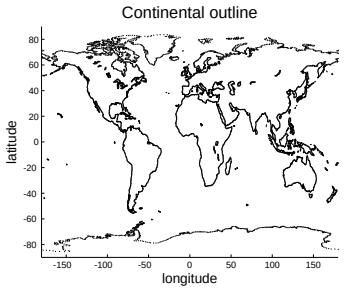

Figure 2.3: Continental outline (of sorts).

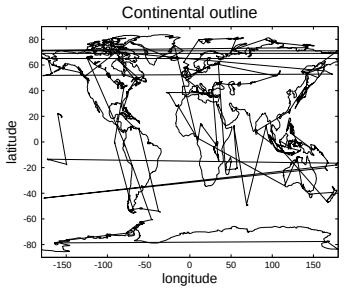

Figure 2.4: Another continental outline (of sorts).

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 have 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)[55]. 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[56] and hence loop through all the line fragments. Yes? For instance, setting `n=1`, and plot (with `n` replacing

`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).

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

[56] 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 ...

`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[57])?

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 2.5 appears(?)

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

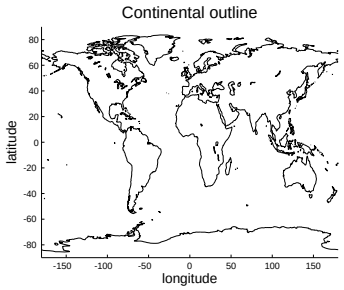

Figure 2.5: Another go at the continental outline!

3
Further ... MATLAB and data visualization

This chapter is something of a potpourri of MATLAB data and visualization methodologies and techniques, generally building on the basics covered in Chapter 2.

3.1 *Further data input*

Previously, you imported ASCII data into **MATLAB** using the `load` command[1]. You might not have realized it at the time, but the use of `load` requires that your data is in a fairly precise format. **MATLAB** says "*ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, %).*" Firstly, your data may not be in a simple format and often may contain both numerical values and string values. Secondly, your data may not even be in a text/ASCII format. For instance, you data maybe be in an Excel spreadsheet, or for spatial scientific data, an increasingly common format is called 'netCDF' (Network Common Data Form). In this section, we'll go through the basics and some examples of each.

[1] Or maybe 'cheated' and used the **MATLAB** GUI ...

3.1.1 *Formatted text (ASCII) input*

The general procedure that you need to follow to input formatted text data is as follows:

1. First, you need to 'open' the file – the command (function) for this is called `fopen` (see Box). You need to assign the results of this function to a variable for later use.
   What is going on and why this all differs so much form using load, where you only had to use a single command, is that you first have to open a connection to the file ... before you even read any of the contents in(!)[2].
2. Secondly ... you can read the content in (finally!). The complications here include specifying the format of the data you are going to read in. You also need to tell **MATLAB** the ID of the file that you have opened (so it knows which one to read from). The function you are going to use to do this is called `textscan`.
3. Close the file using `fclose` (see Box). You are going to have to pass the ID of the open file again when you call this function (so MATLAB knows which file to close).
4. Lastly, you are going to have to deal with the special data structure that MATLAB has created for you ...

If you are interested (probably not) – the connection made to an open file is called a file *pipe*. Typically, you have have multiple open file *pipes* at the same time in programs, and this is why obtaining and then specifying a unique ID for the *pipe* you wish to read or write through, is critical.

opening and closing files

MATLAB has a pair of commands for opening and closing files for read/write:

- `fopen` will open a file. It needs to be passed the name (and path if necessary) of the file (as a string), and will return an ID for the file (assign (save) this to a variable – you'll need it!).
- `fclose` ... will close the file. It requires the ID of the file (i.e. the variable name you assigned the result of calling `fopen` to) passed to it as a parameter.

`textscan`

According to (actually, paraphrased from) **MATLAB**:

```
C = textscan(ID,format)
```

" *... reads data from an open text file into a cell array,* `C`*. The text file is indicated by the file identifier,* `ID`*. Use* `fopen` *to open the file and obtain the* `ID` *value. When you finish reading from a file, close the file by calling* `fclose(ID)`."

The `ID` part should be straightforward (if not – follow through the Example).

The `format` bit is the complicated bit ... There is some help in a following Box and via the Example. Otherwise, there is a great deal of details and examples in **MATLAB help** – you could look at this as a sort of menu of possibilities, and given a particular file import problem, the best thing to do is simply scan through help, looking for something that matches (or is close to) your particular data problem (and/or ask Google).

[2] This is very common across all(?) programming languages.

As an initial Example to illustrate this alternative (and more flexible) means of importing of data, we are going to return to the paleo atmospheric CO_2 proxy dataset file – `paleo_CO2_data.txt`[3]. Assuming that you have already (previously) downloaded it, open it up in a text editor and view it – you should see 4 neatly (ish) aligned columns of numeric values ... and nothing else[4].

OK – so having seen the format of the data in the ASCII file, you are going to work through the following steps[5]:

1. First 'open' the file – you will be using the function command `fopen`, and passing it the filename[6] (including the path to the file if necessary). So that you can easily refer to the file that you have opened later, assign the output of `fopen`[7] to a variable, e.g.

```
» openfile_id = fopen('paleo_CO2_data.txt');
```

2. Now ... this is where it gets a trickier – the function you are going to use now is called `textscan`. Refer to **help** on `textscan`, but as a useful minimum, you need to pass 3 pieces of information:

   (a) The ID of the open file (you have assigned this to a handy variable (`openfile_id`) already.)

   (b) The *format* of the file (see margin note). (This is where it gets much less fun, but hang in there!) You simply list, space-separated, and between a single set of quotation marks, one format option per element of data.

   In this particular Example, there are 4 items of data (per row) – each of them is an integer or a floating point number[8], depending on how you want to look at it. Assuming that the data is a floating point number, the *format* for the input of each number item, is `%f`.

The result of `textscan` is then assigned to a parameter, e.g.

```
my_data = textscan(openfile_id,'%f %f %f %f');
```

3. So far, so good! And you can now close the file:

```
» fclose(openfile_id);
```

4. Actually, it does get worse before the end of the tunnel ... what `textscan` actually returns, i.e. your read-in data, is placed into an odd structure call a *cell array*. It is not worth our while worrying about just what the heck this is, and if you view it in the **Variables** window (i.e. double click on the `cell array` name in the **Workspace** window), it does not display the simple table of 4 columns of data that maybe you were expecting. For now, we can transform this format into something that we are more familiar with using the `cell2mat` function, e.g.

[3] The version that you have used before – not to be confused with a version ending in `.dat` that we will look at shortly ...

[4] This 'nothing else' is important as it is the reason why you were previously able just to `load` the data.

[5] You can start off working at the command line if you wish, but ultimately, you are going to need to put everything into an **m-file**.

[6] For convenience, you could assign the filename (+ its path) to a (string) variable and then simply pass the variable name – remember, no `' '` needed for a variable naming containing a string (whereas `' '` is needed for the string itself).

[7] The output is a simple integer index, whose value is specific to the file that you have opened.

According to **MATLAB help**:
"*the `format` is a string of conversion specifiers enclosed in single quotation marks. The number of specifiers determines the number of cells in the cell array C.*" Take this to mean that you need one format specifier, per column of data. The specifier will differ whether the data element is a number or character (and MATLAB will further enable you to create specific numerical types).

The format specifiers are all listed under `help textscan`. However, your Dummies Guide to `textscan` (and good for most common applications) is that the following options exist:

```
%d - (signed)integer
%f - floating point number
%s - string
```

MATLAB will automatically repeat the format for as many lines as there are of data. Alternatively you can specify precisely how many times you would like the format repeated (and hence data read in).

[8] At least, none of them are clearly strings, right?

```
my_data_array = cell2mat(my_data);
```

And now ... it is done, i.e. there exists a simple array, of 4 columns, the first being the age (Ma), the second being the CO_2 concentration value (units of ppm), and the 3rd and 4th; minimum ad maximum error estimates in the proxy reconstructed value. :)

As a further example, we are going to process a more complicated version of the paleo atmospheric CO_2 proxy dataset. The file is called `paleo_CO2_data.dat` and is available from the course webpage. An initial problem here is even opening up the file to view it – if you use standard **Windows** editors such as **Notepad** it fails to format it properly when displaying its contents[9]. The first lesson then in scientific computing then is to have access to a more powerful/flexible editor than default/built-in programs such as **Notepad**. One good (**Windows**) alternative is **Notepad++**[10]. So go open the file with this instead[11]. Note the format – there are a bunch of header lines and moreover, some of the columns are not numbers (but rather strings). So even if you were to edit out the headers with `comments` (`%`)[12], you are still left with the problem of mis-matched columns. You could edit the file in **Excel** to remove the problematic columns ... but now this seems like a real waste of time to be editing data formats with one software package just to get it into a second! (Again, you could use the **MATLAB** GUI import functionality ... but it will be a healthy life experience for you to do it at the command line :o))

OK – so having gotten an idea of the format of the ASCII data file, you are going to work again through the 4 steps:

1. First 'open' the file as before (`fopen`) and assigned the ID returned by the function to a variable `openfile_id2`.
2. Call `textscan`. However, we now want to pass 3 pieces of information (compared to 2 before):
   (a) The ID of the open file.
   (b) The *format* of the data.
   (c) And now – a parameter, together with an (integer) value, to specify how many rows of the file, assumed to be the header information, to skip.

(Again – the result of `textscan` is then assigned to a variable which will represent a *cell array*.)
Lets do the easy bit first – to tell **MATLAB** to skip n lines of a file, you add the parameter `'HeaderLines'` to the list of parameters passed to `textscan`, and then simply tell it how many lines to skip. In this Example, you'd add:

MATLAB claims that a `cell array` is "*A cell array is a data type with indexed data containers called cells. Each cell can contain any type of data. Cell arrays commonly contain pieces of text, combinations of text and numbers from spreadsheets or text files, or numeric arrays of different sizes.*" I am sort of prepared to believe this.

Basically, in object-oriented speak, a cell array is an object, or rather, an array of objects. As MATLAB hints – the cells can contain *anything*. Your limitation previously is that an array had to be all floating point numbers, all integers, or all strings, and if strings, all the strings had to be the same size. For strings in particular, it is obvious that a more flexible format where a vector could contain both 'banana' and 'kiwi' is needed (try creating a 2-element vector with these 2 words and see what happens). You clearly might also want to link a number with a string (e.g. number of bananas) in the same array, rather than have to create 2 sperate arrays.

`cell2mat`

Having created this weird format (`cell array`), now MATLAB has to give you a way of converting the data inside into something more usable. The function is `cell2mat`, which for a cell array C:

```
A = cell2mat(C);
```

will return the corresponding ('normal') array A.

Now this is only true if all the data in `C` is of the same tpye (e.g. all floating point numbers). If the data types are mixed or you only wish for a sub-set of the data to be extracted and converted, simply index the required part of the cell array (Examples on this later).

[9] If you use a **Mac** (or **linux**) however, all text editors should display the content jus fine.

[10] Conveniently installed on the Watkins computer lab computers.

[11] Right-mouse-button-click over the file, then select **Open with** and then click on **Notepadd++**.

[12] Recall that **MATLAB** ignore lines starting with a `%` and this includes loading in data lines using `load`.

```
my_data = textscan(openfile_id2, ...  ,'HeaderLines',3);
```

OK – now to dive back into the MATLAB syntax mire ... Lets just load in just the first 2 columns of data, and assume that they are both integers (and skipping the first 3 lines of the file as per above). We might guess that we could simply write:

```
my_data = textscan(openfile_id2,'%d %d','HeaderLines',3);
```

Try it (including closing the file, and a call to `cell2mat`, as before). What has happened?
It seems that MATLAB translates your format (`'%d,%d'`) into: 'read in a pair of integers, and keep automatically repeating this, until something else is encountered'. That something else is sequence of characters at the end of the first data line (line #4, because we skipped the first 3), that makes **MATLAB** think that it has finished (or rather, that it cannot reading in 2 pairs of integers any longer). This leaves you with 2 pairs of integers – i.e. a 2×2 matrix (as you'll see if you look at `my_data_array`).
Here is a solution – we could omit all the information following the first 2 elements (something for Google to help with).[13]:

```
my_data = ...
     textscan(openfile_id2,'%d %d %*[^\n]','Headerlines',3)
```

3. Now close the file:

```
fclose(openfile_id);
```

4. And now convert the results to something more human-readable:

```
my_data_array = cell2mat(my_data);
```

This should do it – a simple array, of 2 columns, the first being the age (Ma) and the second the CO_2 concentration value (units of ppm). :)

There must be some sort of important life lesson hidden here. Perhaps about only working with well-behaved data files, or using the GUI import functionality? But hopefully it does illustrate that messy files can be dealt with, without the need for laborious editing or processing in **Excel**.

3.1.2 Importing ... Excel spreadsheets

If your data is contained in an Excel spreadsheet, and you want it in **MATLAB**, your options are:

1. Select some, or all, of the columns and rows in a specific worksheet, and either copy-paste this into a text file (but taking care that the worksheet column widths are formatted such that they

[13] This turns out to be specifying `'%*[^\n]'`, which in effects sort of says:
'skip everything (all the fields) (`%*`) up until the end of the line is found (`[^\n]`).

are wider than the widest data element), or save in an ASCII format, with comma or tab delineations between columns. In either case, then load in the data using `load`, or if consisting of mixed numbers/text, go through the Hell that is `textscan` ....
2. Use **MATLAB** function `xlsread`.

So ... option #2 looks ... is looking the easiest ... :)

As an example, lets return to the paleo proxy CO_2 data again, but this time, as an Excel sheet. The data file you need is:
paleo_CO2_data.xlsx
(You may as well go load this into Excel just to take a look at the format and so subsequently, you'll know if you have imported it faithfully or not.)

From the help box on `xlsread`, it should be pretty apparent what you do. And in fact, I am going to leave you to work it out – try and import the age and CO_2 data (the numeric part of the data) from **paleo_CO2_data.xlsx**.

If you need to, you index a cell array, pretty well much like a normal array, except it has an alternative syntax. For a normal, numeric array `A`, you might write:

```
» A(4,3)
```

to reference the value in the 4th row, 3rd column. For a *cell array* `C`, to index the cell in the 4th row, 3rd column, you'd also write:

```
» C(4,3)
```

but you'd get a cell returned, not the value in the cell. If you want the value in the cell located at `(4,3)`, you'd put the index in curly brackets:

```
» C{4,3}
```

and you'd get a value of `3000` returned in the example of `raw`.

`xlsread`

There are various uses (i.e. alternative allowed syntax) for `xlsread` for an **Excel** file with name `filename`. The 2 relevant and more useful ones look to be:

1. `num = xlsread(filename)` which will return the *numeric* data in the Excel file `filename` in the form of a matrix, `num`. Note that non-numeric (e.g. string) headers and/or columns, are ignored. Also note that `num` is a 'normal' numeric array and does not require any conversion.
2. `[num,txt,raw] = ... xlsread(filename)` will additionally return text data in a *cell array* `txt`, and *everything* in a cell array `raw`.

You can also specify a particular worksheet out of an Excel file to load in:

```
num = ...
xlsread(filename,sheet)
```

(and there are further refinements and options listed under **help**).

3.1.3 *Importing ... netCDF format data*

Much of spatial, and particularly model-generated, scientific output, is in the form of *netCDF* files. This is a format designed as a common standard to facilitate sharing and transfer of spatial data, but in a way that enables e.g. a 'complete' description of dimensions and various types of meta-data to be incorporated along with the data. The format is platform independent and a variety of graphical viewers exist for viewing and interrogating the data. Most programming languages support the reading and writing of *netCDf* format data. **MATLAB** is no exception here.

As per the previous subsection on data import, and indeed file read/write in programming languages in general – one opens a file and receives an ID for that file. The file can then be written to or read (including just interrogating its properties rather than necessarily extracting spatial data) using this ID. And of course, closed (using the ID). However, the *netCDF* standard is a little odd in how reading/writing is implemented and everything has to be done by determining the ID of a particular data variable or property of the file. As you'll see ...

The general approach for reading *netCDF* data is as follows:

1. Open the netCDF file by

```
ncid = netcdf.open(filename,'nowrite');
```

where `filename` is the name of the *netCDF* file (which generally will end in `.nc`). `'nowrite'` simply tells **MATLAB** that this file is being open as read-only (this is the 'safe' option and prevents accidental deletion of over-writing of data).

2. This is the weird bit, as we cannot ask for the data we want automatically :o) Instead, given that we know[14] the name of the variable we want to access, we ask for its ID ...

```
varid = netcdf.inqVarID(ncid,NAME);
```

where NAME is the name of the variable (as a string), allowing us to then request the data:

```
data = netcdf.getVar(ncid,varid);
```

that says – assign the data represented by the variable `varid`, in the *netCDF* file with ID `ncid`, to the variable `data`.

So actually, not totally weird – you request the ID of the variable, then use that to get access to the data itself. The names of the **MATLAB** commands vaguely make sense in this respect – `inqVarID` for inquiring about the ID of a variable, and `getVar` for getting the variable (data) itself[15].

3. Finally – close the file, by passing the ID variable into the function `netcdf.close`, i.e.

```
netcdf.close(ncid);
```

Note that you need to pass the ID of the *netCDF* file for each and every command (after `netcdf.open`) so MATLAB knows which *netCDF* object you are referring to.

[14] There are ways of listing the variables if not.

[15] It is beyond the scope of this course to worry about why in the case of netCDF, the function are all `netcdf.` something. Just to say, it involves objects and methods and is a common notation in object orientated languages (that nominally, **MATLAB** isn't).

For a *netCDF* Example, we'll take the output of a low resolution Earth system model (GENIE). To start off, download the '2D marine sediment results' *netCDF* file – **fields_sedgem_2d.nc**. The data here

is relatively simple – a 2D distribution of bottom-water and surface sediment properties, saved at a single point in time. In other words, there are only 2 (spatial) dimensions to the data[16].

OK – we'll start by opening the file (assuming that you have downloaded it!), remembering to assign its unique ID to some variable. Then, you'll want to get hold of (and assign to another variable), the ID of the variable we want to get hold of and plot – in this Example, it is called `'grid_topo'`. Having obtained the ID for this variable, you can then fetch it – assign it to a variable `data`. Then close the file.[17]

You should now have an array called `data`. It should be 36×36 in size. Why not plot it[18]. Can you guess what it might be? Is it in the correct orientation? (If not – correct it.)

Clearly what is missing are the x and y axis values, which you should have correctly deduced are longitude and latitude, respectively, with latitude presumably going from -90 to 90N, and longitude ... well, maybe it is not completely obvious exactly what the value of longitude is at the original.

A great strength of *netCDF* its the ability of this file format to also contain the grid (axis) details that the data is on. There are ways of finding out the names of the axis variables (dimensions), but for now, I'll give you them:

- `'lat'` – is the latitude axis. (Technically, the axis values are the mid-points of the grid cells.)
- `'lon'` – is the longitude axis.

The axes are held in the *netCDF* file as vectors and we can retrieve this (1D) data in a similar way to the 2D data:

```
varid = netcdf.inqVarID(ncid,'lat');
lat   = netcdf.getVar(ncid,varid);
varid = netcdf.inqVarID(ncid,'lon');
lon   = netcdf.getVar(ncid,varid);
```

in which we obtain the ID of the axis variable `'lat'`, then retrieve the axis data and assign it to a vector `lat` (and then likewise for longitude). Do this, and confirm that you get plausible vectors representing positions along a longitude and latitude axis.

The final task would then be to take the 2 axis vectors, and create a pair of matrices – one containing longitude values associated with the 2D data points, and one containing latitude values associated with the 2D data points. For this, you need to use `meshgrid`[19]. See if you can create the necessary lon/lat matrices and then plot the model topo data on its correct axes.[20]

The variable names of other data-sets that you might load and experiment with in terms of plotting function, color scale, and any

[16] Adding time would make it 3 dimensions (2 spatial + 1 of time). Adding height or depth in the ocean would also make it 3 (3 spatial). 3 spatial + time would make for a 4-dimensional dataset ...

[17] You should be able to do all of this without further hints – the sequence of commands and how they are used, is given in the introduction to this subsection.

[18] Your choice of 2D plotting function.

[19] See subsequent section.

[20] If you have flipped the data matrix around earlier when plotting, un-do this, or re-load the 2D data, or else the axes will no long correspond to the data matrix orientation ...

other refinements that help visualise the data, include:

- 'ocn_sal' – deep ocean salinity (units of per mil).
- 'ocn_O2' – concentration of oxygen in bottom waters (units of mol kg^{-1}).
- 'sed_CaCO3' – % of calcium carbonate in surface sediments.

In a related *netCDF* Example, we'll extend the problem to 3D – 2 spatial dimensions (longitude and latitude) and one of time. The file you need is called **fields_biogem_2d.nc**[21]. You are going to go through the same basic procedures of: opening the file, obtaining the variable ID, accessing the data using that ID, and closing the file. The name of the variable is called `'atm_temp'`. Create a script to do this all, calling the data array that you obtain by calling

```
netcdf.getVar(ncid,varid);
```

`data3`. How many dimensions does this array have? What are the lengths along each dimension? Can you guess which dimension of the 3 time is?

The name of the time axis variable is `'time'`, and you can access the times along this axis (i.e. the times at which the model saved a 2D spatial state) by:

```
varid = netcdf.inqVarID(ncid,'time');
times = netcdf.getVar(ncid,varid);
```

Ideally, you should be able, given the 3D array that you have obtained (from the data variable `atm_temp`), to specify and plot, the 1st model-projected surface air temperature distribution, as well as the last distribution. And given that the variables for latitude and longitude are also `'lat'` and `'lon'`, you should be able to plot the temperature distribution with appropriate axes (and contoured).

You should also ... using `find`, be able to determine (and plot) the 2D data slice corresponding to the year (mid point) 1999.5.

Finally, test yourself and understanding to date, by creating an animation of how the surface air temperature in the model evolves over time.[22]

[21] The back-story is that this contains the 2D surface ocean and atmosphere fields form a model experiment in which the climate system was spun-up from rest and uniform values of everything, so as time progresses, the spatial patterns of the climate system start to evolve and stabilize.

[22] You have everything you need – the vector of times, and from this you can determine how many times there are and hence the number of iterations of a loop.

3.2 *Further (spatial / (x,y,z)) plotting*

As you have seen earlier – the simplest possible way of taking a matrix of data values and plotting them spatially, as a function of (*x*,*y*) location, is the function `image`. In effect, this is treating your data as if it were an image – the data values being the 'color' of each pixel and the location in the matrix defining where in the image (row, column) the pixel is. The problem with this is that information regrading what is on the *x* and *y* axes is lost, be this distance, lat/lon, or some set of observed/experimental variables, or whatever. instead, the points are evenly spaced on both axes. Moreover, the raw values are plotted and there is no possibility of interpolation/contouring or smoothing. One could regard scatter plotting as an improvement over this and a sort of *x*,*y*,*z* plotting, in as much as a 3rd dimension (*z* data value) can be represented through color and/or symbol shape and at time this can be quite effective. However, again, no interpolation/contouring or smoothing is possible with `scatter`.

For plotting true (*x*,*y*,*z*)/'3D' plots (i.e. data values in 2 spatial dimension), MATLAB provides a wide variety of more formal ways of plotting data spatially, including even the possibility of adding a 4th dimension representing the data value (*x*,*y*,*z*,*zz*) (see Box).

For a feel of what you should be able to learn to achieve using **MATLAB** – go to the following webpage. In this data repository you can do things like re-plot with different longitude, latitude, and temperature ranges. Overlay the coastlines, and other useful things like that. You can also click through the different months of the year to get a feel for how the surface temperatures on Earth change with the seasons. Note that the graphic produced from this particular website is not particularly great, and you can all do better than this using **MATLAB** already. Presumably there are some lazy PhD students out there lacking the skills that you are (hopefully) learning. Perhaps they should take GEO111 (or maybe you are ...)?

x,y,z PLOTTING

MATLAB calls plots of a (*z*) value as a function of both *x* and *y*, '3D'. Strictly, one could look at some of these functions as 2D, as scatter can plot a 3rd data (*z*) value as different colors/shapes/sizes as a function of both *x* and *y* ... Anyway, the most commonly used/useful and fortunately simple, functions which create a 2D (*x*, *y*) plot but with contours in the value of (*z*), are:

1. `contour` – Plots a figure with the data contoured, with a range and increment between contours that is fully specifiable, color-coded or not, and labelled or not. Options are also provided for specifying how the contouring is done (and the data interpolated).
2. `contourf` – Similar to `contour`, except in between the (now simple black, by default) contours, a fill color is plotted and scaled to the data value.

For a genuine 3D plot, with surface height determined by the data in the 3rd dimension of the array, colors and/or contours in the data in the 4th array dimension, suitable functions include:
`surf`, `surfc`, `mesh`
(but are not considered further here).

As an example, load in the global topographic data file (**etopo1deg.dat**) from the course webpage. This is the height of the (solid) surface of the Earth relative to mean sealevel in meters, with the continents having a positive value and the ocean floor, negative. The data is conveniently on a 1° (longitude and latitude) grid. You could view the resulting elements of the 2D array in the Variable window if you like ... but at 360×180 in size, there may not be much of use you can glean by visually inspecting the matrix[23].

Try throwing the array into the `image` function see what happens (hopefully something like Figure 3.1). It it had happened to come out

`imagesc` For a data array (matrix) `A`,

```
imagsc(A)
```

displays the data array as if a bitmap, but unlike `image` (see earlier), *"uses the full range of colors in the colormap*".

[23] More useful then are the summary details in the **Workspace window**, such as the apparent absence of NaNs and that the **Min** and **Max** Earth surface heights seem plausible.

displayed upsidedown[24], then you'd need to flip the matrix upside-down using the command:

```
etopo1deg=flipud(etopo1deg);
```

and if the Earth instead appeared on its side[25], you need to swap the rows and columns (x for y axis):

```
etopo1deg=etopo1deg';
```

It is not unusual for a first plotting attempt of spatial data to be incorrectly orientated and a little trial-and-error to get it straight is perfectly acceptable!

This is not exactly the prettiest of images. You can distinguish ocean (blue) from land (mostly brown, but other color pixels in places). Fortunately, MALTAB provides a variant of this plotting function, `imagesc`, that calculates the color scale to exactly span the min/max values in the data. Try it (and get something like Figure 3.2 hopefully).

The function `imagesc` also enables the range of data values the colo9r range corresponds to, to be set. Refer to `help` on this function and see if you can plot just the above-sealevel, i.e. land surface heights, spanning zero (sealevel) to the maximum height[26].

Which sort of in a round-about sort of way also brings us to how to set the color scale, which can be changed using the `colormap` command (see Box). Try out some different *colormaps* and re-plot the global topography data.What scales work well and what do not? Which scales help pick out details of e.g. ocean floor depth variation and which help pick out simple land-sea contrasts. Think about what one might want to highlight about global topography and what color scale might be best for this purpose?

Sticking with global Earth surface topography, how else can we display the spatial data? For instance we might want to interpolate it, contour it, or simple get the longitude and latitude exes correct. Note that only by luck, because this particular dataset is 1 degree by 1 degree, the default axis scale in MATLAB when using image is approximately correct, although note that 'latitude' has been ordered in reverse and it goes from 1 to 180 rather than -90 to 90 ... We'll explicitly address this shortly.

To start with, you can simply use the `contour` function (see Box), passing only the matrix (of global topography values). Try this. Now you might want to think about flipping the matrix up-down, and/or left-right, as your plot should have come out looking like Figure 3.3.

Once you have fixed the orientation of the topography map, you might play about with the color scale (`colormap`) as before. You

[24] It doesn't in this particular case.

[25] Actually, in this example, it is OK in this respect too. Boring!

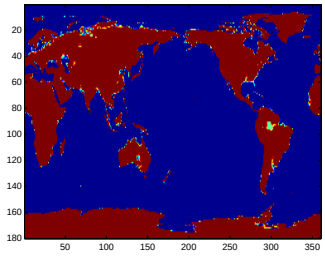

Figure 3.1: Very basic imaging (`image`) of an array (2D) of data – here, global bathymetry.

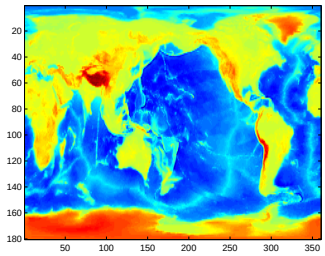

Figure 3.2: Slightly improved very basic imaging (`imagesc`) of bathymetry data.

[26] Don't forget the function `max`.

`colormap` **MATLAB** has a number of 'colormaps' built in – color scale that determine the colors that correspond to the data. The command to change the *colormap* from the default is:

```
» colormap NAME
```

where `NAME` is the name of the *colormap*. You can find a list of possible *colormaps* in **help** on `colormap` (in a table towards the bottom). But a brief summary is:

- `parula` – the current **MATLAB** default – chosen to provide a wide range of color and color intensity.
- `jet` – the old **MATLAB** default, but one which uses red and green in the same color, which should be avoided (why?).
- `hot`, `cool` – relatively simple color transitions but useful – hot is something like you'll see in publication figures.
- `pink` – another simple and at times useful transition and from dark (almost black) to white.

To return to the default *colormap*:

```
» colormap default
```

might also try the companion to `contour` – `contourf`. Re-orientating the matrix, switching to a different *colormap*, and plotting using `contourf`, might give you something like Figure 3.4.

OK, so a next refinement in plotting esp. maps and contour plots, is firstly to specify the range of the color scale, as we may not want the min-to-max range chosen by default by **MATLAB**, and the number of contours (e.g. in the topography example, they are pretty far apart and it is difficult to make out much detail). Both of these factors can be addressed simultaneously, by giving MATLAB a vector containing the value at which you want the contours drawn[27].

Taking the global topography data – lets say you were interested only in low lying and shallow bathymetry, and wanted 20 contours intervals. Assuming a range in topographic height (relative to sealevel) of -1000 m to +1000 m, you should be able to deduce how to create the vector(?)[28]

Do this and check e.g. by opening up the vector in the **Variables window**. You should see the numbers from -1000 to 1000 in intervals of 100. Why, for instance, can you not simply write:

```
» v = [-1000:1000];
```

??? (Or rather: why might this not be a good idea ... ?)

Having created a specific vector of contours to plot, try it out. OK – so this is a little weird and maybe not so useful, but you get the point hopefully. So try plotting the following:

1. Just above sealevel topography, up to 10,000 m, in increments of 100 m.
2. Just the sealevel (coastline) contour ... trickier – create a vector with a value at zero, and a value either side – one very high and one very low. Use `contour` rather than `contourf`, although the latter produces a lovely land-sea mask!
3. Convert the data matrix of value in units of m, to ft, and plot the ocean floor (values equal to or below sealevel) in intervals of 1000 ft.
4. Finally – try some different color scales for the above. Think about which color scales best help illustrate the data, and whether `contour` or `contourf` is clearer. Also: how many contour intervals is 'best'? You key is to make features clear, within the plot becoming cluttered or overly detailed.

The final refinement in contour plotting we'll look at is adding labels to the contours. The command to do this is `clabel` (for 'contour label') (see Box). Now, before anything, there is a slightly complication. `clabel` needs to know details of the contours and graphics object with which to do anything with. For the purposes of this course,

[27] By default: MATLAB determines the minimum and maximum data values, and draws 10 equally spaced contours between these limits.

[28] If not, it is:

```
» v = [-1000:100:1000];
```

`contour` There are various uses of contour. The simplest is:

```
contour(Z)
```

where `Z` is a matrix. Thsi ends up similar to image except with the data contoured rather than plotted as pixels (the 'simularity' here is that the x and y axis values simple are the number of the rows and columns of the data).

You can specify the values at which the contours are drawn, by passing a vector (v) of these values, e.g.

```
contour(X,v)
```

More involved and practical, is:

```
contour(X,Y,Z)
```

where `X`, `Y`, and `Z`, are all matrices of the *same* size (there is important). `X` and `Y` contain the x and y coordinate locations of y data values (contained in matrix `Z`). In the example of a map – `X` and `Y` contain the longitude and latitude values of the data values in `Z`.

Similarly, you can add a vector `v` containing the contours to be drawn, by:

```
contour(X,Y,Z,v)
```

you don't have to worry about the details of this, but simply need to know the following:

1. When you call `contour` (or `contourf`), 2 parameters are returned, which so far you have not cared about or even noticed. We now need them. SO when you call either potting function, using the syntax:

   ```
   [C,h] = contour( ...  )
   ```

   which saves a matrix of data to `C`, and a ID (technically: graphics object 'handle')to `h`.
2. When you call `clabel`, pass these parameters back in, e.g.

   ```
   clabel(C,h)
   ```

(in its most basic usage).

If you do this, in an earlier example of plotting just the zero height contour, and now using the most basic default usage of clabel (as above), you get, for good or for bad, Figure 3.5.

In the default usage of `clabel`, you'll get a label added on every contour that you plot. This ... can get kinda messy if you have lots and lots of contours plotted. You may well not need every single contour labelled, particularly if you also provide a color scale (see below). So you can also pass in a vector to tell MATLAB which contours to label. For example, if you have a contour interval vector:

```
v = [-1000:100:1000];
```

maybe you onyl want labels every 500m, so you'd use a vector:

```
w = [-1000:500:1000];
```

to specific the labelling intervals. The complete set of commands becomes:

```
» v = [-1000:100:1000];
» w = [-1000:500:1000];
» [C,h] = contour(etopo1deg,v);
» clabel(C,h,w);
```

Finally – missing from our color-coded plots so far, is a color scale to relate values to colors (although labelling the contours works as an OK substitute). The MATLAB command is simple:

```
» colorbar
```

(and see Box for further usage). Try adding a *colorbar*, and in different places in the plot. Refer to the Box to try and add a caption to it ...

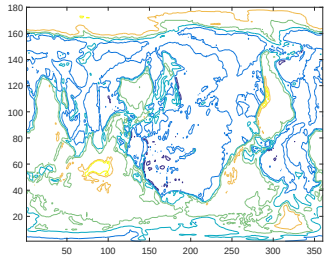

Figure 3.3: Example result of basic usage of the `contour` function.

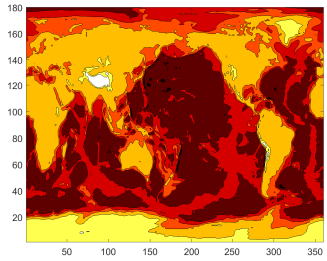

Figure 3.4: Example usage of `contourf`, with the `hot` *colormap* (giving dark-/brown colors as deep ocean, and light/white as high altitude).

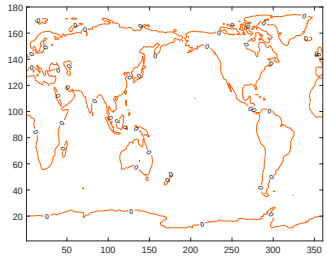

Figure 3.5: Example usage of `contour`, contouring only the zero height isoline, and providing a label.

`clabel`

```
» clabel(C,h)
```

labels every contour plotted from

```
[C,h] = contour( ...  );
```

(or from `contourf`).

By prescribing and passing a vector `v` of contour intervals, you can label fewer/specific intervals rather than all of them (the default), e.g.

```
» clabel(C,h,v)
```

In this next Example, we'll address the issue with missing/incorrect lon/lat axis labels on the plots.

Each data point in the `etopo1deg` matrix should have one longitude value (x-axis) and one latitude (y-axis) value associated with it. It should hopefully be intuitive to you now ... that what we need is a pair of matrices, of exactly the same size as the `etopo1deg` data matrix – one holding longitude values and one latitude values. There are various ways of creating the required matrices 'by hand' (or involving writing a program including a *loop*). All of them are tedious. There is a **MATLAB** function to help. But it is not entirely intuitive[29] ... `meshgrid`.

Spend a few minutes reading about it in `help`. In particular, look at the examples given to help you translate the **MATLAB**-speak gobbledegook of the function **Description**. You should be able to glean from all this that this function allows us to create two $a \times b$ arrays; one with the columns all having the same values, and one with the rows all having the same values (exactly what we need for defining the (lon,lat) of all the global data points). If not, and probably not – see Box. And then lets do a simple example (adapted from **help**):

```
» [X,Y] = meshgrid(1:3,10:14)
X =
     1  2  3
     1  2  3
     1  2  3
     1  2  3
     1  2  3
Y =
     10  10  10
     11  11  11
     12  12  12
     13  13  13
     14  14  14
```

Here, we are taking 2 vectors – `[1:3]` and `[10:14]`, and asking **MATLAB** (very nicely) to create 2 matrixes, one in which `[1:3]` is replicated down, until it has the same number of rows as the length of `[10:14]`, and one in which `[10:14]` is replicated across until it has the same number of columns as the length of `[1:3]`. (Try it.)

It'll become apparent *why* bother shortly. Honest.

In our Example – start my noting that the topography data is on a regular 1 degree grid starting at 0° longitude. Latitude starts (at the bottom) at -90° and goes up to +90°). We need a matrix containing all the longitude values from 0° to 359° and latitude from -90° to 89°

`colorbar`

This almost could not be simpler:

```
» colorbar
```

plots the color scale! By default, is places it to the RH sice of the plot. If you wish for it to appear anywhere else, use the modified syntax:

```
» colorbar(PLACEMENT)
```

where PLACEMENT is one of: `'northoutside'`, `'southoutside'`, `'eastoutside'`, `'westoutside'`. Note that these are strings and so need to be in quotation marks. (More options are summarized in a table in **help**.)

Finally, you can also add a label to the *colorbar*, but only if you get hold of its ID ('graphics handle') when you call `colorbar`, e.g.

```
» h = colorbar
```

will save the graphics handle in variable `h`, which you can then muck about with via:

```
c.Label.String = 'The
units of my lovely
colorbar';
```

(Don't fight this – use this syntax to set a label for the *colorbar* – don't worry about what it means. **MATLAB** keeps rather annoyingly changing the way it does this anyway :()

[29] **DON'T PANIC!**

`meshgrid`

The unholy syntax is:

```
[X,Y] = meshgrid(xv,yv)
```

Pause, and take a deep breath. On the left – the results of `meshgrid` are being returned to 2 matrixes, `X` and `Y`. These are going to be our matrixes of the longitude and latitude values (in the particular example in the text). So far so good(?)

On the right, passed into the function `meshgrid`, are two vectors – `xv` and `yv`. Pause again.

What **MATLAB** is going to do, is to take the (row) vector `xv`, and it is going to replicate it down so that there are as many rows as in the vector `yv`. This becomes the returned output matrix `X`. **MATLAB** then takes the column vector `yv`, and replicates it across so that there are as many columns as in the vector `xv`. This becomes the returned output matrix `Y`.

.[30] These matrices need to be the same size as the data matrix.

Maybe just 'do' it and then understand what has happebned after. Create the longitude and latitude grids by:

```
» [lon lat] = meshgrid([0:359],[-90:89]);
```

View (in the **Variables window**) the `lon` matrix first. Scan through it. Hopefully ... you'll note that it is 360 columns across, and in each column has the same value – the longitude. The matrix is 180 rows 'high', so that there is a longitude value for each latitude. Similarly, view `lat`. This also should make a little sense if you pause and think about it, with the one exception that the South Pole latitude is at the 'top' of the matrix – don't worry about this for now ...

The only way to fuly make sense of things now, is to use it. Remember that use of `contour` (and `contourf`) can take matrices of x and y (here: longitude and latitude) values that correspond to the data entries in the data matrix (`etopo1deg`). Re-load the topography data in case you have flipped it about in all sotes of odd ways, and then do:

```
» [lon lat] = meshgrid([0:359],[-90:89]);
» contour(lon,lat,etopo1deg);
```

Almost! Note that the x and y axis labelling is 'correct' and particaulrly the y-axis, where latitude gos from -90 to 90 (although by default **MATLAB** labels in intervals of 20 starting at -80 it seems). But it also turns out that we do need to flip the data op-side-down. We can actually do this in the same line as we plot:

```
» contour(lon,lat,flipud(etopo1deg));
```

Phew! (Figure Figure 3.6.)

The final complication is that the data points in the gridded dataset (matrix `etopo1deg`), technically correspond to the mid-points of a 1 degree grid, not the corners. So if we were going to try and be formally correct[31], our vectors that we'd pass into `meshgrid`, would be:

```
» xv = [0.5:359.5];
» yv = [-89.5:89.5];
```

[30] There is a slight complication with this, which we'll get to shortly, but note that the data array is 360 elements (x-direction) by 180 elements (y-direction).

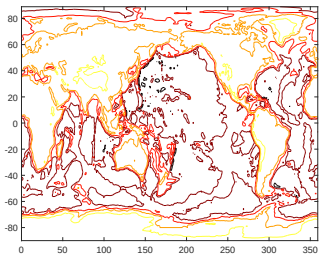

Figure 3.6: Usage of `contour` but with lon/lat values created by `meshgrid` function and passed in (and with the `hot` *colormap* (giving dark/brown colors as deep ocean, and light/white as high altitude).

[31] Don't worry about this for now – grids will be covered more in subsequence chapters surrounding numerical (environmental) models.

OK – ANOTHER EXAMPLE on this. Previously, you downloaded and plotted monthly global distributions of surface air temperature. You plotted these simply using `pcolor` (or `image`) and the results were ... variable. Certainly not publication-quality graphics and missing appropriate longitude and latitude axes for the plots.

Make a copy of your original *script* (**m-file**) in which you created the animation, and give it a new name. Edit your program, and in place of `pcolor`, use `contour` or `contourf` (your choice!). Pass in just the data matrix (of monthly temperature) when calling the `contour(f)` function and don't yet worry about the lon/lat values. Get this working (i.e. debug it if not). You should end up with a contoured animation (rather than a bit-map animation).

The problem with the axis labelling should be much more apparent (than compared to the topography data, which was on a handy 1 degree grid already). So you need to make a matrix of longitude values, and one of latitude. using `meshgrid`. The grid is a little awkward:

1. The longitude grid runs from 0°E (column #1) with an increment of 1.875°; i.e., 0.000°E, 1.875°E, 3.750°E, ... up to 358.125°E (column #192).
2. Latitude runs from 88.54196°S (-88.54196°N) at row #1, to 88.54196°N (row #94) with an increment of about 1.904.

so I'll give you the answer up-front:

```
» lonv = [(1.875/2):1.875:360-(1.875/2)];
» latv = [-90+(1.904/2):1.904:90-(1.904/2)];
» [lon lat] = meshgrid(lonv,latv);
```

Now use the longitude and latitude values matrices, in conjunction with `contour(f)`, to plot the global temperature distributions 'properly'. Try plotting just one plot first, before looping through all 12 months.

At this point (before creating an animation), you might also explore some of the plotting refinements we saw earlier. For example, as per Figure 3.7. Firstly – get the units of the temperature data array into units of °C or °F rather than °K. Either: assign the `temp` array data to a new array and make the appropriate conversion from °K (all within the loop), or you can do this subtraction on the line that you actually plot the data (i.e., within the `contour/contourf` function), for example:

```
contourf(lon(:,:),lat(:,:),temp(:,:,month)-273.15);
```

would convert to °C as it plotted the data.

You can also get the plotting temperature limits and contouring consistent between months and with greater resolution by adding the following line (before the loop starts):

```
v=[-40:2:40];
```

and then to the `contour(...)` (or `contourf(...)`) function, add `,v` to the end of the list of passed parameters. This particular choice for the vector `v` tells MATLAB to do the contouring from -40 to 40 (°C), and

at a contour interval of 2 (°C).. Play around with the min and max limits of the range, and also with the contour interval to see what gives the clearest and least cluttered plot. For instance, maybe you don't want the low temperatures to go 'off' the scale (the white color in the filled contour plot).

3.2.1 *Plotting maps*

You can do some nice spatial plotting with this data using the **MATLAB Mapping Toolbox**. This should be available as part of the **MATLAB** installation in the Lab (and also if you have downloaded and installed an academic version on a personal laptop). Refer to the online documentation for the **Mapping Toolbox** to get you started. The key function appears to be `geoshow`. Try plotting the region encompassing the 'quake data, with a coastal outline (of land masses), and the 'quake data overlain. Explore different map projections. Remember to always ensure appropriate labelling of plots.

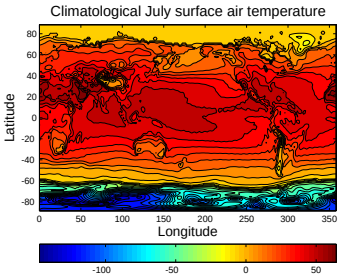

Figure 3.7: Example contour plot including `meshgrid`-generated lon/lat values. Result of `contourf(lon,lat,temp7,30)`, where the data file was `temp7.tsv`, with some embellishments.

4
Further ... Programming

In this chapter we'll get some (more) practice building programs and crafting (often) bite-sized chunks of code that solve a specific, normally computational or numerical (rather than scientific) problem (*algorithms*) [1].

[1] According to the all-mighty Wikipdeia (and who am I to argue?) – an "algorithm ... is a self-contained step-by-step set of operations to be performed. Algorithms perform calculation, data processing, and/or automated reasoning tasks."

4.1 find!

So – a single **MATLAB** function gets a high-level section, all to itself. Either it's really powerful and useful, or I am running out of ideas for the text[2].

`find` ... finds where-ever in an array, a specific condition is met. If the specific condition occurs once, a single array location is returned. The specific condition could occur multiple times, in which case `find` will report back multiple positions in the array.

What do I mean by a 'specific condition'? Basically – exactly as per in the `if ...` construction – a conditional statement being evaluated to true.

OK – some initial Examples.

Lets say that you have a vector of numbers, e.g.:

```
A = [3 7 5 1 9 7 4 2];
```

and you want to find the maximum value in the vector – easy[3]

But ... you want to find *where* in the vector the maximum value occurs. Why might you want to do this? Rarely do you have a single vector of data on its own – generally it is always linked to at least one other vector (often time or length in scientific examples). Trivially, our second vector might be:

```
B = [0:7];
```

and is time in ms. The question then becomes: at what time did the maximum value occur? Obviously, this is easy by eye with just 8 numbers, but if you had 1000s ...

We can start by determining the maximum value.

```
c = max(A);
```

Now, we use `find` to evaluate where in the array `A` (here: a vector) the element with a value of `max(A)` occurs, or where the condition `d == c` is true,where `d` is the element in question (the maximum value). So:

```
find(A(:)  == c);
```

should do it. Here, what we are saying is: take all of the elements in A and find where an element occurs that is equal to `c` (the maximum value which we already determined). Try it, and MATLAB should return `5` – the 5th element in the vector.

Finally, if we assign the result of find to `d`, we can then use `d` to determine the time at which the value of `9` occurred, i.e. `B(d)` which evaluates to `4` (ms):

In this example, `find` returned just a single element, but if we instead had:

[2] It is really powerful and useful.

`find`
MATLAB defines find, with a basic syntax of:

```
k = find(X)
```

as 'return[ing] a vector containing the linear indices of each nonzero element in array X'. That means ... nothing to me. This is going to have to be a job for some Examples ... (in order to see what `find` is all about).

[3] I hope so ... check back earlier in the course on `max`.

```
A = [3 9 5 1 9 7 4 2];
```

The maximum value is still the same (9) but now ...

```
» find(A(:)  == c)
ans =
    2
    5
```

What has happened is that `find` has determined that there are `2` elements in vector `A` that satisfy the condition of being equal to `c` (9) and these lie at positions (index) `2` and `5`. The result vector, if you assigned it to the variable `d` again, can be used just as before to access the corresponding times in vector `B`;

```
» d = find(A(:)  == c); » B(d)
ans =
    1 4
```

i.e. that the times at which the values of `9` occur are `1` and `4` (ms).

Any of the relational operators (that evaluate to *true* or *false*) can be used. In fact – looking at it this way leads us to maybe understand the **MATLAB help** text, because *true* and false*a* are equivalent to `1` and `0`, and `find` is defined as a function that returns the indices of the non-zero elements in a vector. By writing `A(:)  == c` we are in effect creating a vector of `1`s and `0`s depending on whether the equality is *true* or not for each element. You can pick apart what is going on and see that this is the case, by typing:

```
» A(:)  == c
ans =
    0
    1
    0
    0
    1
    0
    0
    0
```

(the statement being *true* at positions (index) `2` and `5`, which is exactly what `find` told you).

For instance, we could ask `find` to tell us which elements of A have a value greater than 5:

```
» find(A(:)  > 5)
ans =
    2
    5
    6
```

(Inspect the contents of vector `A` and satisfy yourself that this is the case.)

We can also use `find` to filter data. Perhaps you do not want values over 5 in the dataset. Perhaps this is above the maximum reliable range of the instrument that generated them. Having obtained a vector of locations of these values, e.g.

```
d = find(A(:)  > 5);
```

we can plug this vector back into `A` and assign arrays of zero size to these locations – effectively, deleting the locations in the array, i.e.

```
A(d) = [];
```

They it, and note that the size[4] of `A` has shrunk to `5` – all the other elements remain, and in order, but the elements with a value greater than 5 have gone. You could apply an identical deletion (filtering) to the time array (`B(d) = []`).

[4] Use the command `length` or view in the **Workspace** Window.

Play about with some other relational operators and criteria, and make up some vectors of your own until you are comfortable with using `find`.

Back to the 'quake Example: Find[5] how may earth quakes there were bigger than M = 8? Also determine how many quakes occurred bigger than M = 7, 6, 5, 4, and 3. Determine the day on which the magnitude 8.7 shock occurred.

[5] Intentional joke *and* clue.

In the first problem (number of quakes greater than a specified limit) – you need ask find to return the row numbers for all quakes satisfying the condition: `magnitude > 8.0`. `find` will return you a column vector. You don't actually need to worry about or access the contents of the vector, you just need to know how many elements there are in the vector (because there will be one element for each occurrence of `magnitude > 8.0`). This is the same as its `length` (see earlier and/or **help**).

In the second problem – you need to find the row number of the quake magnitude data which satisfies the condition: `magnitude > 8.7`. Knowing the row number, you can then access the data column containing the sate information, and hence extract the day and solve the problem.

All these problems can actually be solved in a single line of **MATLAB**, but feel free to break it down into multiple steps.

In the sealevel (oxygen isotope) Example, you could start by determining the maximum and minimum sea-levels that have occurred

over the last 782,000 years. Then ... because it would be helpful to know *when* the minimum and maximum sea-level heights occurred, use the `find` function to find the data row in which the minimum and maximum values occurred. Once you know the respective data rows, you can then easily pull out the ages.[6] Find the ages of both minimum and maximum values.

Also find all the occasions (times) on which sealevel was higher than today (modern). (Or equivalently, when the oxygen isotope value, that we are assuming directly reflects changes in level, was lower than modern[7].)

You can also ask questions based in time, such as what was the sealevel (or oxygen isotope value) at 21 ka (i.e. without having to look through the data manually and determine on which row 21 ka occurs, because this is exactly what `find` can do this for you)? This can be particularly useful if the value of time is calculated or passed in from elsewhere, rather than specified as e.g. 21 ka, because you may not *a priori* know what the value will be, hence automating the script with `find` is super useful. Effectively then you are creating an *algorithm* for taking a time input and determining sealevel.

[6] HINT – if your maximum value was stored in the variable max_value, you found find the corresponding row by: `find(data(:,2) == max_value)`

What this is saying, is search the 2nd column (the sea-level values) of the array `data`, and look for a match to the value of `max_value`. The equality operator (`==`) is used in this context.

[7] Lower d18O => less ice volume => higher sealevel.

For an Example of data-filtering – dig out the paleo-proxy (not ice-core) atmospheric CO_2 data you downloaded. One further way of plotting with `scatter` is to scale the point size by a data value. We could do with by:

```
SCATTER(data(:,1),data(:,2),data(:,2))
```

... except ... it turns out that there are atmospheric CO_2 values of zero or less and you cannot have an area (size) value of zero or less ...

This leads us to a new use for `find` and some basic data filtering. The simplest thing you could do to ensure no zero values, would be to add a very small number to all the values. This would defeat the 'no zero' parameter restriction, but would not help if there were negative values and you have now slightly modified and distorted the data which is not very scientific. Substituting a `NaN` for problem values is a useful trick, as MATLAB will simply ignore and not attempt to plot such values.

So first, lets replace any zero in the CO_2 column of the data with a `NaN`. The compact version of the command you need is:

```
data(find(data(:,2)==0),2)=NaN;
```

But as ever – perhaps break this down into separate steps and use additional arrays to store the results of intermediate steps, if it makes it easier to understand, e.g.

`NaN`

... is **Not-a-Number** and is a representation for something that cannot be represented as a number, although if you try and divide something by zero **MATLAB** reports `Inf` rather than a `NaN`.

`NaN` can also be used as a function to generate arrays of `NaN`s. The most common/usage in this context is:

```
N = NaN(sz1,...,szN)
```

which will (according to **help**) "generate a a sz1-by-...-by-szN array of NaN values where sz1,...,szN indicates the size of each dimension. For example, `NaN(3,4)` returns a 3-by-4 array of NaN values."

```
list_of_zero_locations = find(data(:,2)==0);
data(list_of_zero_locations,2) = NaN;
```

What this is saying is: first find all the locations (rows) in the 2nd column of `data` which are equivalent (==) to zero. Set the CO_2 value in all these rows, to a `NaN` (technically speaking: assign a value of `NaN` to these locations). You have now filtered out zeros, and replaced the offending values with a `NaN` and when **MATLAB** encounters NaNs in plotting – it ignores them and omits that row of data from the plot.

Alternatively, we could have simply deleted the entire row containing each offending zero. Breaking it down, this is similar to before in that you start by identifying the row numbers of were zeros appear in the 2nd column, but now we set the entire row to be 'empty', represented by `[]`:

```
list_of_zero_locations = find(data(:,2)==0);
data(list_of_zero_locations,:)  = [];
```

If you check the **Workspace window**[8], you should notice that the size of the array data has been reduced (by 4 rows, which was the number of times a zero appeared in the 2nd column).

We are almost there with this example except it turns out that there is a CO_2 proxy data value less than zero(!!!) We can filter this out, just as for zeros. I'll leave this as an exercise for you[9] ... The plot should end up looking like Figure 4.1. As another lesson-ette, given that the circles are insanely large ... try plotting this with proportionally smaller circles[10].

As a last (optional) exercise on this ... In the CO_2 data, there are min and max uncertainty limit values. One could color-code the points in a scatter-plot to represent either the min or the max (perhaps try this first), but one on its own is not necessarily much use. One could color-code by the difference, but this is a function of the absolute value and one would expect large uncertainty bars if the mean (central) estimate was high, and lower if it were low. Perhaps we need the *relative* range in uncertainty? Can you do this? i.e., `scatter`-plot the mean CO_2 estimate (as a function of time), but color-coding for the range in uncertainty as a proportion of the value?

It turns out this is not entirely trivial because as you have seen, the data is not as well behaved as you might have hoped. In fact, it is just like real data you might encounter all the time! Before you do anything – break down into small steps what you need to do with the data, as this will inform what (if any) additional processing you might have to carry out on the data. It should be obvious, that to create a CO_2 difference, *relative* to the mean, you are going to have to divide by the mean value (column #2 in the array). So first off –

[8] Or:

```
» size(data)
```

[9] But you might e.g. use <=.

[10] HINT: you are going to want to apply a scaling factor to the vector you passed as the point size data.

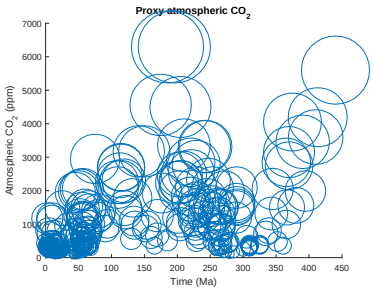

Figure 4.1: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

if any of the mean values are zero, it is all going to go pear-shaped. Actually, equally unhelpful, or at least, lacking in any meaning, may be negative values. If you inspect the data (in the **Variable window**), there are both zeros and negative values for mean CO_2 proxy estimates. We need to get rid of these. Follow the steps as before. You may also have to process the min and max values should they turn out to be the same. Likely you are going to have to delete all the rows in which (1) column #2 values are zero or below, and (2) column #3 and #4 values are equal (you could also try the `NaN` substitution and see if it works out). (If you need a slight hint ... one possible answer is here[11] , but try and work it out for yourself.)

All that is missing now, is any indication of what the color scale actually means in terms of values (and of what). MATLAB will add a colorbar to a plot with the command ... `colorbar`. Although the color scale gets automatically plotted with labels for the values, looking at the plot, we still don't know what the values are of (e.g. units). We can label the colorbar, but MATLAB needs to know what we are labelling. Each graphic object is assigned a unique ID when you create them and which normally you know nothing about. We can create a variable to store the ID, and then pass this ID to MATLAB to tell it to create a title for the colorbar. To cut a long story short:

```
colorbar_id=colorbar;
title(colorbar_id,'Relative error (%)';
```

It should end up looking something like Figure 4.2 in which you can see the high relative uncertainty (bight colors) prevail at low CO_2 values and 'deeper time' (ca. 200-300 Ma). The colorbar title (label) is maybe not ideal, nicer would be one aligned vertically rather than horizontally. We'll worry about that sort of refinement another time.

[11] In this possible solution – all rows in the array `data`, with mean CO2 values less than or equal to zero, are deleted. Also, all rows for which the max and min values are the same, are also deleted.

```
» data=load('paleo_CO2_data.txt',
...'-ascii');
» data(find(data(:,2)<=0),:)=[];
» data(find(data(:,3)==data(:,4)),:)
...=[];
» scatter(data(:,1),data(:,2),40,
...100*(data(:,4)-data(:,3))./data(:,2),
...'filled');
» xlabel('Time (Ma)')
» ylabel('Atmospheric CO_2 (ppm)')
» title('Proxy atmospheric CO_2')
```

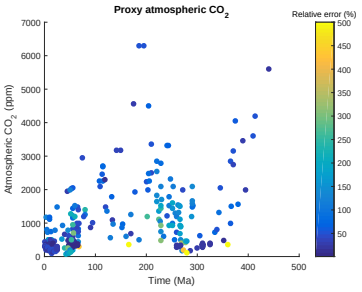

Figure 4.2: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

5
Graphical User Interfaces (GUIs)

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

5.1 *MATLAB GUI basics*

MATLAB kindly[1] provides a tool (itself a GUI) for creating GUIs – the 'Graphical User Interface Development Environment' (**GUIDE**). **GUIDE** does 2 main things for you:

1. Firstly, it facilitates the design of the GUI window(s).
2. Secondly, it creates a code framework for the associated program.

You run **GUIDE** at the command line by typing its name:

```
» guide
```

and a window as shown in Figure 5.1 should appear. We'll only concern ourselves with the default option amongst the (4) 'GUIDE templates' – 'Blank GUI (default)'[2]. As for the tick-box 'Save new figure as:' – we'll leave this alone[3]. The 'Preview' window is blank at this point because you have selected a blank template (d'uh!) (and are not loading in a previously created GUI). So, all that remains, is to go ahead and click on 'OK'.

Actually ... before you move on, it is worth pausing at this point and reflecting on what happened and what the implications are for what you might like to do (GUI-wise). At the command line, you entered the command `guide`, which presumably ran a script or function (a piece of code in any case). A windows (the 'GUIDE Quick Start' window) was summoned (actually a figure window was created). The (figure) window did not open completely blank, but instead you might not:

- It has a close/minimize/maximize buttons at the top right (and the window can be re-sized).
- It has a title at the top (in the title bar) with a cute (barf) **MATLAB** icon.
- There are 3 buttons at the bottom right – 'OK', 'Cancel', and 'Help'. Presumably they'll all do something (different) when clicked.
- Everything else is neatly enclosed in a pair of *tabs* (one labelled 'Create New GUI' and one 'Open Existing GUI' and you can switch between tabs by clicking on the required tab.
- In the 'Open Existing GUI' tab, there is:
  - A list (of template names plus that annoying cute little icon again).
  - An area with a border labelled 'Preview' with a grey box labelled 'Blank' in the middle.
  - There is a *tick box* and next to it (grey-ed out by default), a box with a file path and name in and to the right of that, a

[1] For once, it is not a sperate, zillion-dollar license ...

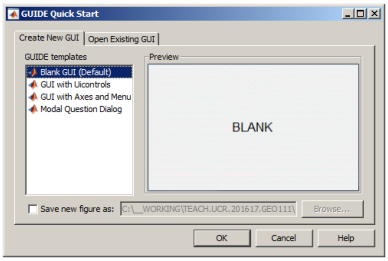

Figure 5.1: Starting GUI window of the **MATLAB GUIDE**, GUI design tool.

[2] So don't go randomly clicking on anything just yet!

[3] You can save the resulting figure (and code) under whatever filename you wish, later anyway. (If you really want, you can enter it in now here – it makes little difference.)

button labelled 'Browse'.

In essence, most of the primary (or at least, basic) features of a GUI are here to see. Funnily enough, nothing much had changed, at least in Windows, since ... the 80s[4]. Maybe that is a good thing as despite the **MATLAB GUIDE** tool being completely new to you, you hopefully can guess what would generally likely happen if you clicked on random bits of the 'GUIDE Quick Start' window.

(If you have not already clicked OK – do it now.)

Rather than creating some basic code first[5], MATLAB now throws you straight into a window design tool as per Figure 5.2. There is a lot going on here, but start by noting there is the usual drop-down menu bar at the very top (under the title bar ('untitled.fig') of the window) and a row of icon underneath that (no re-appearance of the **MATLAB** icon thankfully). At the bottom of the window there is some information, mostly about location (of what?). To the left of the window is a group of icons[6] plus a (depressed, by default) mouse pointer icon. Most of the window is made up of a pane (whose contents apparently is, or might be, larger than the area shown as indicated by the presence of scroll bars along the right and bottom edges). The pane itself is ruled with a grid pattern.

Again – the great advantage of familiarity (of program GUI design) – you might guess (you'd be correct if you did) that the icons to the left allow you to select an object and place it in the pane, the grid serving to help you position the object. And this leads us to an important point – creating GUI-based programs is as much (or more) about design as it is about programming. The cleverest program (and most complex calculations) might simply be a total fail if the GUI is wholly unappealing or complete un-intuitive (or lacks a GUI entirely). The grid is hence there for a reason and that is to guide you towards creating an ordered (and aligned), logical, and uncluttered arrangement of things (we'll come to what the 'things' are shortly) within the GUI window.

You might be tempted ... to click on everything and throw all sort of objects (what things?) into the pane of your embryonic GUI window. But the more GUI objects you have ... ultimately, the more code and the more debugging[7] you'll have to do. So we'll start as simply as possible and build up.

[4] That is: 1980s, as much as some might believe **Microsoft** has made little progress since the 1880s ...

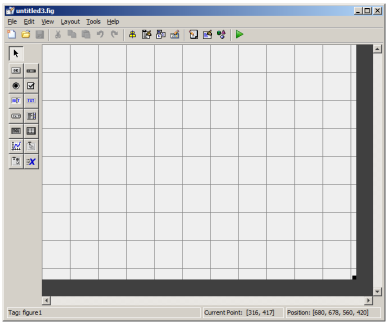

Figure 5.2: (Blank) GUI window editor GUI window.

[5] Actually, **MATLAB** has done this too and you would have seen it open up in the **Code Editor** window if you have provided a filename in the 'GUIDE Quick Start' window.

[6] Still no re-appearance of the MATLAB icon!

[7] Which has a steep power relationship with the amount of code.

5.1.1 *Hello, World [Static Text (box)]*

This is as simple as it is going to get for a 'program' with a GUI. In the GUIDE window editor already open, if you haven't fatally mucked about with it, or open up a new GUI by typing `guide` (or `GUIDE`) at the command line again – identify the Static Text icon (by

hovering the mouse pointed over an icon, its function is revealed). Click (L mouse button) on it. The mouse pointer, when over the gridded design pane, should change to a cross-hairs.[8] Find a convenient place .... perhaps at the intersection of two grid lines, click the mouse down and drag out a box – this will be the size (and location) of the Static Text object. Release the mouse button to finish. If you don't like the size or location, you can move/re-size just like you would to a **Windows** (or **MacOS** etc.) window.

So far, the (static) text object as a rather unappealing content of 'Static Text' in a pretty small font. You can edit the properties of this object by double-clicking on it[9]. Whoa! That's a long list of ... actually, *properties* of the object. Each property (the column on the left) has a default value (the column on the right) assigned to it. Evidently, you can edit the properties using the design tool rather than in the code code, setting a parameter value.[10] For now, we'll just make two changes:

1. For the String property – click in the box to the right, delete 'Static Text' and write 'Hello, World'.
2. The text is pretty small ... so for the FontSize property, click in the box to the right, delete 8.0 and write ... well, try something larger.

Within reason, play with some of the other properties if you like (at least, the ones that you can make a reasonably informed guess as ot what they do). Maybe you end up with a design window looking like Figure 5.3. Note that the effect of your changes is only shown if you e.g. hit Enter or click on a different property. If you accidently click outside of the text object an in the design pane, you'll end up switching the property editor to the window itself, which you don't want. (You can simply click back inside the text object to return the property editor to the text object's settings.)[11]

When you are done (editing properties) – click the Save icon. If this is a GUI that you have not previously created or previously assigned a filename to, you'll get a Save As dialogue box. At this point, **MATLAB** is going to save the window design with a .fig extension.

Something a little scary now happens – **MATLAB** opens up the code editor and some code, with a filename the same as you entered in but now with a .m extension. There is nothing we need worry about ... yet. And in fact, half the file is taken up with a main function that has the comment: `DO NOT EDIT`. Please take this advice ... :o)

Close the design window (and the code editor if it distracts you). At the command line, type the filename (no extension) to run the automatically generated code **m-file**. A window opens up ... the

[8] Note that this is to facilitate the positioning of the icon rather than being anything about guns and shooting at the coders behind **Windows**.

[9] I didn't actually read this anywhere – the operation of the editor or Windows has the same feel and intuitive usage as the sort (hopefully) of Windows you you are going to create in your GUI(s)).

[10] In reality: **MATLAB** is secretely writing the relevant code and setting the parameter value ...

[11] Unfortunately, the title of the property editor window is completely unhelpful – matlab.ui.control.UIControl when the text object properties are being edited, and matlab.ui.Figure when the (figure) window properties are being edited. So maybe watch out for Figure appearing in the title bar as an indicator or quite what is being edited.

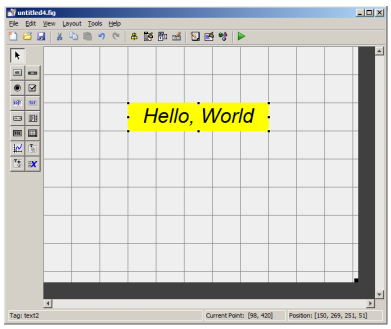

Figure 5.3: Design of the Hello, World window!

contents should come as no surprise, because you have just specified them (via the GUIDE GUI design tool). Your first GUI! But one you might notice does not actually 'do' anything – it just sits there unresponsive. Although you can at least close it (because it is automatically generated with the usual basic close/minimize icons plus the name of the m-file in the *titlebar*.

5.1.2 *Simple GUI responses [Push Button]*

A GUI is only of any particular use if it allows some response to input. This is going to involve a little code ... so we'll start with the simplest possible action – a *button* that performs a simple action (closes the window).

Re-run `guide` and open up a new window editor (by clicking OK in the GUIDE Quick Start window). Now find the Push Button icon, click it, and drag out a push button object in the design pane. You should see a box (with a pseudo 3D shading at the edges) with the text Push Button in the centre as per Figure 5.4. As before, you can edit the properties of the push button object (because the default properties are totally boring) by double-clicking it. Start by editing the font (size) and message. Perhaps 'Go away!'. And save ...

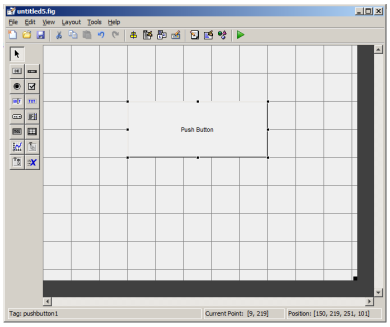

Figure 5.4: Design window with a default push button object.

When it saves, MATLAB again opens up the code it generated. There is slightly more code in the file this time and shortly, we'll need to look at it. But for now: type the name of the file at the command line. You'll get a window opening with the push button you created in it. Click on it. It does seems to 'respond' (pretends to depress by means of changing the edges with the pseudo 3-D shading) to the mouse click, but ... nothing else happens. This is where YOU (and your amazing coding skills) now come in.

If you have closed the design window, re-run `guide` and rather than creating a new GUI – switch to the Open Existing GUI tab and double-click your filename (of the push button GUI) or select and OK. Double-click on the push button object to open up the property editor. We'll make only one (more) change here – down the list of properties your fine 'Tag'. This is the name (ID or *handle*) of the push button object.[12] By default, the name is pushbutton1. Edit this to ... goawayButton (or pick an alternative name) and re-save the GUI.

[12] In essence, no different from a filename – a unique identifier for an object (/file).

Go to the code editor for the associated **m-file** (which will have the same name remember). In the file we have:

- The main function which we can ignore (and indeed apparently should not be edited!).
- `function goaway_OpeningFcn` which is executed when the GUI is started up. This is the place to put code for initializing models or whatever.

- I have no idea what `function varargout = goaway_OutputFcn`. Textbooks helpfully say to ignore this. Great idea.
- Finally, `function goawayButton_Callback`. This function is executed when your 'Go Away!' push button is pressed.

In this simple GUI, we have only one figure and it is active (has the mouses' attention), so we could simply use the `close` command ('*deletes the current figure*'). Insert this simple command in the `function goawayButton_Callback` function, after the last comment line.[13] Save the **m-file** and re-run. Now if you click on the 'Go Away!' push button, the window does indeed go away (aka, closes).

[13] Note that automatically generated MATLAB code does not seem to ever formally `end` a function as one really should do ...

5.1.3 *Updating object properties (do you like bananas?)*

Bananas. Do you like them? Perhaps the GUI can provide an answer (rather than just text statements written to the command line via `disp` as before).

Now you are going to want to think about the design of the GUI a little. What we want is for the the GUI to display a question ('Do you like bananas?'). There will be two options, 'Yes' and 'No' that can be clicked. Depending on which one is clicked, some appropriately supportive, or otherwise, message will appear in response. We need:

1. A plain (static) *text box* as before to display the question.
2. A pair of *push buttons* (again as before).
3. Another plain (static) *text box* to display the answer/response.

And ... we are going to need some code that, depending on which button is pushed, displays a different message.

The latter part is not as bad as it sounds. We could have no test initially in the 2nd (static) *text box*. We just need to change its text property (i.e. change the no text to our message). This is mostly a case of working out and using the unique identifier of this text box object AND the identifier of the text property (of the text box object).

Firstly – re-run GUIDE. Create a new GUI window with the 4 elements (2 static text boxes and 2 push buttons). It is up to you how you arrange these 4 objects in the design pane. You might be guided how windows in programs you have used are designed. AT the minimum, it is standard practice to place a 'No' push button next to and aligned horizontally, with the 'Yes' (and often 'Yes' to the right of 'No').

No idiot would design anything like Figure 5.5 and certainly not with those color choices ... but you get the idea.

For each of the objects (2 text boxes and 2 push buttons), I have renamed them (the Tag property) to something more memorable than e.g. button or box, #1, #2, #3, etc etc..

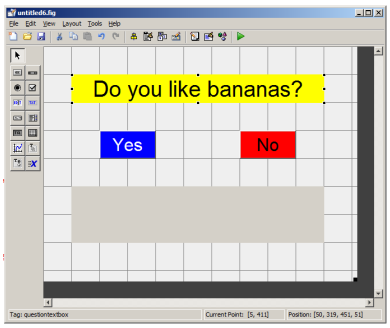

Figure 5.5: (completely) Bananas design window.

The code that MATLAB generates for `bananas.m` (my name) is not a lot more involved than before. Primarily, there is just a second function associated with a mouse click on the 2nd push button.

The logic is going to be very simple. In fact, we don't need any, because if the Yes button is clicked, **MATLAB** will call one function (my name: `function yesbutton_Callback`), and if the No button is clicked, the other function (`function nobutton_Callback`) is called. As alluded to above, how do we get the text to change in the 2nd text box (from the default of no text)?

Unfortunately, **MATLAB** get all weird here.[14] If you had a friend called Luna, you might reasonably communicate with them via the name 'Luna'. MATLAB doesn't do it this way and instead assigns a numeric ID. Luna might have an ID of 8.206034. So you are going to have to get this ID, which in this case is the ID of the 2nd text box, if you want to change a property (here: the displayed text).

[14] Actually, no wierder than **netCDF**. Or arguably **Python** ...

First off, you can get the ID of the object property using `findobj` and assign the result to some memorably variable, e.g.

```
h_answertext = findobj('Tag','answertextbox');
```

This is as simple(!) as asking to find the ID of the object which has a `Tag` with value 'answertextbox' (which was the value I set in the design editor).[15]

[15] What we might refer to as an ID, **MATLAB** calls a *handle*. Hence commonly an 'h' might appear at the start of a variable name to indicate it contains a *handle*.

Where would we put this line of code? Why not in the initialization function, `function bananas_OpeningFcn` and I am guessing, at the end of that function[16].

[16] Be careful as MATLAB is not automatically adding an `end` to the end of functions ...

Now – we have the ID of the 2nd text box and we can now set its property (from no text t a suitable message). Lets first implement an answer if the Yes push button is clicked. The command to set a property is ... `set`. In our example, the handle we have already obtained and assigned to the variable `h_answertext`. The name of the property we want to change (refer to the column list in the property editor if you like as a reminder) is `'String'`. And the text ... well, you can have whatever you want. The complete line is then:

```
set(h_answertext,'String','Yes, it is an excellent fruit.');
```

`set`
Sets ... the property value of an object. The syntax is:

```
set(h,name,value)
```

where `h` is the handle (the ID obtained via `findobj`), `name`, is the name of a property, and `value`, the value of a property.

Well, it turns out that this does not quite work? Why? My guess that we could add the line:

```
h_answertext = findobj('Tag','answertextbox');
```

to the initialization function was incorrect. One possibility is that the function states:

```
% -- Executes just before bananas is made visible.
```

meaning that this function is called before the window is opened. If the objects have not yet been created at this point, they will obviously

have no ID and the variable `h_answertext` would be empty. Alternatively, the variable `h_answertext` created in the initialixation function is simply not accessible (visible) to `function yesbutton_Callback`. Sod it, the line has been added to `functionyesbutton_Callback` instead, giving:

```
h_answertext = findobj('Tag','answertextbox');
set(h_answertext,'String','Yes, it is an excellent fruit.');
```

Now it works and creates the result shown in Figure 5.6.

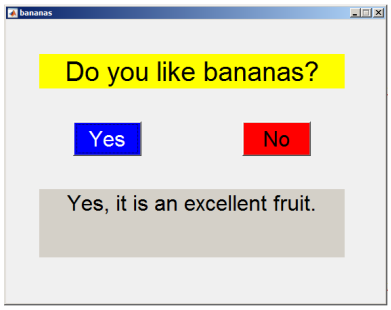

Figure 5.6: (completely) Bananas GUI in action.

Now extend this so that an alternative answer is provided if the 'No' button is instead clicked. Other embellishments you could make might be to make the color of the button you clicked change. This is simply a matter of finding its object ID, and setting the property BackgroundColor.

Finally, and to put a little of your coding skills to the test, how about displaying a 3rd message ('Make up your mind!'?) if someone changes their mind – i.e. if a second button is pressed (after the first). You'll need a variable to store whether any button has been pressed and assign this an initial value of false, e.g.

```
var_pressed = false;
```

Whenever a button is pressed, `var_pressed` will become (will be set to) true. So before displaying the message in both of the button press *callback* functions, the value of `var_pressed` needs to be tested – a false means this is the first time any button has been pressed. Once that initial message is displayed, the `var_pressed` becomes true, and when the next time a button is pressed and the value of `var_pressed` tested, a *true* leads to a different message. All that is needed is an `if ...` in each callback function, and a line initializing `var_pressed` to false (in `function bananas_OpeningFcn`). There is just one problem ...

Variables in functions are 'sectret' (*private*) and limited (in *scope*) to just that function. So the variable `var_pressed` which you initialized at the end of `function bananas_OpeningFcn` cannot be seen by the callback function.

We can enforce that the same variable is seen by multiple functions by stating that it is *global* (in *scope*):

```
global var_pressed;
```

This line needs to appear at the start of each function in which you need to read or write the value of `var_pressed`, i.e. in both callback functions as well as the initialization function. The complete code for the Yes button call box function would then look like:

```
% -- Executes on button press in yesbutton.
function yesbutton_Callback(hObject, eventdata, handles)
% hObject handle to yesbutton (see GCBO)
% eventdata reserved - to be defined in a future version of
MATLAB
% handles structure with handles and user data (see GUIDATA)
%h_answertext = findobj('Tag','answertextbox');
global var_pressed;
h_answertext = findobj('Tag','answertextbox');
if ~var_pressed
    set(h_answertext,'String','Yes, it is an excellent fruit.');
else
    set(h_answertext,'String','Make up your mind!');
end
var_pressed = true;
```

5.2 *GUI Pokemon game*

Now we'll build on your excellent GUI skills and create a GUI interface for the ballistics (ball trajectory) model.

The idea of the 'game' is that you are going to launch a ball, the behaviour of which will be calculated as per your time-stepping ballistics model. Rather than simply detect whether or not the ball falls below zero (height), there will be a graphic (Pokemon) displayed and a 'hit' will be recorded if the position of the ball falls within the boundary of the graphic. The key initial conditions – initial speed and angle of the launched ball, will be set by controls in the GUI rather than set in code. Finally, there will be a series of refinements to improve the look and feel (and game-play) of the game that will introduce a few further concepts in creating good MATLAB GUIs and also new MATLAB functions. Ultimately, the GUI (app) might look something like Figure 5.7, but how the controls are positioned in the window and their relative size and shape, is pretty well much up to you. You could also control how the initial parameter values are set in a different way (e.g. using an **Edit Text** box rather than a **Slider**). Quite what buttons you want and how they are used is also a matter of personal aesthetics.

There is quite a lot of coding to be done and the risk of a huge mess ensuing. So we'll go through this all in a number of discrete steps:

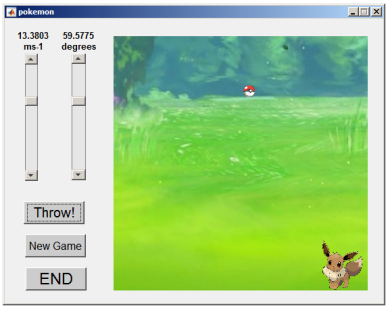

Figure 5.7: Screen-shot of he Pokemon game App.

Part I Create a basic GUI interface using **MATLAB** `guide`.

Part II Load in and display the graphics needed for the game.

Part III Add in the ballistics model.

Part IV Utilizing the sliders.

Part V Create the detection (logic) needed for a successful 'catch' and associated outcomes.

Part VI Refinements to improve the look and feel of the game.

Because of the complexity of the project, the complete code (**m-file**) as well as associated `.fig` GUI file, are provided (on the course webpage). These are provided if needed for guidance (e.g. what code goes where?), only. Try your best to work through the creation of the App without this.

Example images are provided (download via the course webpage) and you can substitute your own if you prefer.

If you run into unexpected and apparently nonsensical 'issues' when you make changes and text the App, try closing the design window and any open Figure windows and type `» clear all`.

Part I – the basic GUI.

To achieve a GUI along the lines of Figure 5.7 you need to create the following objects in the window design editor (but don't create them quite yet – details will follow ...):

1. Something to display all the action and graphics in. This is pretty well much like **MATLAB** creates when you use `plot`, `scatter`, or any of the graphical functions that create a **Figure Window**. This is called an **Axes** object.
2. A **Push Button** for telling MATLAB to start calculating (and displaying) the balls' trajectory.
3. A **Push Button** for resetting the game once it is finished.[17]
4. A **Push Button** to finish the game and close the App.
5. A **Slider** (bar) to set the initial speed of the ball.
6. A **Slider** to set the initial angle of the balls' trajectory.
7. For each slider bar: a **Static text box** to display the value.
8. Also for each slider bar: a **Static text box** to display the units.

[17] This we'll only worry about making use of this in Part IV.

Make a start by running **GUIDE** at the command line. Create a new (blank) GUI. You might save it once the GUI editor window has open up[18]. **MATLAB** then opens the **Editor** and the GUI code template.

[18] **File – Save As...**

Sketch out on a piece of paper how you might lay out the objects in your GUI window before you actually start to create anything. If you have graph paper to hand, you could sketch out your design on a grid similar to the design window grid and size. Note that should should be aiming to make the **Axes** object square (i.e. the same length in both x and y dimension) as the background image we are going to use is square.[19] Also note that the **Sliders** can be horizontal rather than vertical if you prefer and if it make it easier to pack in all the objects.

[19] Later on you might want to try substituting your own background image. In this situation, you might need a different aspect ratio to the **Axes** object.

OK – to begin for real.

1. You have to start somewhere (i.e. you have to pick on one object as the first one to be created!), and the best place to start is arguably with the **Axes** object as it is the largest object in your window. Click on the **Axes** icon and drag out the position and size of the object you want.[20] By default, it is assigned a name (its **Tag** property) of **axes1**. You are not going to have so desperately many objects that it is necessarily worth re-naming it, but you can if you wish (although the text will refer to **axes1** where needed). Remember that you can move and re-size it at any point after creating it. Its position as x,y of the objects origin as well as dimensions (x-length and y-height) are indicated by **Position** at the bottom right

[20] Note that you can drag the GUI editor window larger, and you can also drag larger the gridded design area, meaning that your App window will be larger that you run the program.

of the design window. For e.g. creating an approximately square **Axes** object, you can also simply count the number of grid lines in each dimension.

Save the **.fig** file and run it[21]. You do indeed have a graph-like object with labelled axes. This is not actually that convenient (to have the axes labels when you don't need any in this particular example). In the design window – double click on the **Axes** object to bring up its list of properties. Find and edit **XTick** – delete all the tick mark numbers. Do the same for the y-axis. Close the GUI window from the previous version if it is still open, then save and re-run. Now you should see a large white square(ish) with two thin black lines delineating the axes[22], and nothing else.

[21] Note that there are two things that potentially might both need being saved – the **m-file** and the **.fig** file. If you make code changes, save the **m-file**, and if you make design change sin the GUI editor, save the **.fig** file.

[22] We could remove these black lines, but they'll get covered up later.

2. Next *Push Button* #1. Create (position and size, where- and how-ever you think best). Simplest is to leave the default name ('**pushbutton**1'). Change the text associated with the *Push Button* (property '**String**'). Label as 'Throw', 'Go', or whatever seems appropriate. Remember that you can change the default font size, family, color ... (and e.g. make bold etc.) as well as the color of the button itself (plus a host of other property options).

3. Create a 2nd *Push Button* ('**pushbutton2**') as per before. Label consistent with the GUI aim (and e.g. Figure 5.7).

4. Similarly, create 3rd *Push Button* ('**pushbutton**3').

5. Now we need a **Slider**[23] bar. These are bar with a slider ('knob') that can be slide up and down via the mouse, or moved by clicking in the bar above or below the position of the slider. By doing so (changing the position of the slider along the slider bar), you change the numerical value of the slider. We are going to use one in order to set the initial speed of the ball. So go create one (leaving the default name of '**slider1**').

[23] Not anything to do with baseball.

Because we need to link the **Slider** to our model (in terms of parameter value), we need to specify a minimum and maximum value that the **Slider** can take, as well as an initial value. These properties can be set at in the code, but we'll start off by specifying them using the design GUI tool. If you double click on the **Slider** you'll get its property list opened up. The minimum and maximum property value name are Min and Max – edit these to span a plausible initial speed range[24]. Also set a default initial value (parameter name '**Value**')[25].

[24] I used 0 to $20ms^{-1}$.

[25] I assumed $0ms^{-1}$.

6. Create a second **Slider** ('**slider2**') for setting the initial angle of the ball (*theta*).[26]

[26] Here I assumed a range of 0 to 90°, with a default of 0°.

7. Because the **Sliders** themselves do not tell you quite what value you have slide the slider to, it is a Good Idea to somewhere

display the value. We'll do this via a **Static Text** box ('**text1**') and you'll need to create one to go with each **Slider** (so you'll also have a '**text2**' named object). For now – simply leave the default text property as it is.

8. Finally, if you follow the design in Figure 5.7, you could add a further pair of **Static Text** boxes in order to display the units. This is far from essential and I'll leave it up to you whether you bother, particularly if your window is cluttered already.

That is the basic GUI design done. Save and run (having first closed any open, running, instances of your GUI program). You should have a window with all the objects discussed, but with none of them yet doing anything.

At this point it is worth quickly orientating you around the automatically-generated code **m-file**:

- At the very top:

```
function varargout = pokemon(varargin)
```

appears at the very top of the **m-file** and defines the main function. In this example, the main function is called `pokemon` (meaning the App is run by typing » `pokemon`). Remember that you do not have to edit any of this function.

- Next comes:

```
% -- Executes just before pokemon is made visible.
function pokemon_OpeningFcn(hObject, eventdata, handles,
varargin)
```

This is the function that is called just before the window is made visible and we'll edit it later in order to carry out some initial tasks (i.e. before the ballistics model itself runs).

- Then:

```
% -- Outputs from this function are returned to the command
line.
function varargout = pokemon_OutputFcn(hObject, eventdata,
handles)
```

which is mysteriously useless and we will not edit.

- The first actually useful automatically generated code is:

```
% -- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
```

This will contain the code that is executed when the 'Throw' (or 'Go') button ('**bushbutton1**') is pressed and will end up containing the complete ballistics model code.

- The function code for when second button ('**bushbutton2**') is pressed appears in order after the function associated with '**bushbutton1**':

```
% -- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)
```

We'll only make use of this towards the very end of this section is making the final refinements to the App.

- Then, the third button ('**bushbutton3**'):

```
% -- Executes on button press in pushbutton3.
function pushbutton3_Callback(hObject, eventdata, handles)
```

This will contain more more than a command to close the App (as you have programmed previously).

- The code that is called whenever the position of the first slider the appears:

```
% -- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)
```

- This is then followed by a second function associated with **slider1** whose purpose is ... not obvious. Perhaps slider initialization? Regardless, we'll be ignoring the following code:

```
% -- Executes during object creation, after setting all
properties.
function slider1_CreateFcn(hObject, eventdata, handles)
```

- The final code is the pair of functions for the 2nd slider (of which we'll only edit the first function (`slider2_Callback`)):

```
% -- Executes on slider movement.
function slider2_Callback(hObject, eventdata, handles)

% -- Executes during object creation, after setting all
properties.
function slider2_CreateFcn(hObject, eventdata, handles)
```

Before we move on, you could add your fist code to the **m-file** – a close action if you click on the lower of the three **Push Buttons**. Refer to the previous sub-section and example to remind yourself how to do this. You are aiming to have the App window close when you click on **pushbutton3**, whose associated function is called `function pushbutton3_Callback`.

Save the **m-file** and re-run the App by typing its name (e.g. » `pokemon`) and the command line (first closing any already open instances of it). The App window should now close when you click on the third button. In the GUI design editor, edit the 'value' of the **String** property of this **Push Button** so that it has a logical and vaguely meaningful label.

Part II – (graphics) initialization. Note that in this section, all the code will go in `function pokemon_OpeningFcn`, after the (automatically generated) lines:

```
% Choose default command line output for pokemon
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
% UIWAIT makes pokemon wait for user response (see UIRESUME)
% uiwait(handles.figure1);
```

First, we'll read in a background image ('background.jpg' – available for download from the course webpage) and then display it. We'll use the commands `imread` for reading in the graphics format (and converting it into something **MATLAB** prefers) and then `imshow` to display it. The first part is easy enough:

```
img_background = imread('background.jpg');
```

The question then becomes 'where' to display it. You might not think there is even a question in this – in the window! Except ... where in the window? We actually want the background image in the (currently) blank **Axes** area, not just anywhere in the figure window (which also have various button etc. objects positioned in it). We need to find the ID of the **Axes** object and tell MATLAB that is 'where'.[27] We can get the handle (ID) of the Axes object via:

[27] Actually, it may work without worrying about this, but we'll need to be able to specify where to position other images later anyway.

```
h_axes = findobj('Tag','axes1');
```

and then tell MATLAB that this is currently the object to put things in by:

```
axes(h_axes);
```

We then use this handle in the call to `imread`:

```
h_background = imshow(img_background,'Parent',h_axes);
```

The only problem with this is that **MATLAB** may completely fail to scale the image to fit the **Axes**. We'll fix this shortly.

While we're at it (editing this function), we can specify the axis range for plotting the position of the ball in the Axes object, and add a `hold on` for completeness. We may as well then also define the axis ranges (in m) as parameters (that we can use elsewhere).

The complete code (so far), at the end of the automatically generated code in `function pokemon_OpeningFcn`, becomes:

```
% define grid dimensions
x_max = 10.0;
```

```
y_max = 10.0;
% read in background image
img_background = imread('background.jpg');
% set axes suitable for game
axes(h_axes);
axis([0 x_max 0 y_max]);
hold on;
% draw background
h_background = imshow(img_background,'Parent',h_axes, ...
'Xdata',[0 x_max],'Ydata',[0 y_max]);
```

Now, as part of the call to `imshow`, the size and position of the image is explicitly prescribed.

When you run all this, you should get Figure 5.8.

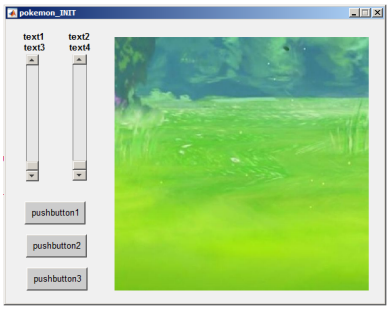

Figure 5.8: Template App with background image.

Next, we want a Pokemon to throw the ball at! The load-in code (which can go after the code fragment above) for the image is identical to before:

```
img_eevee = imread('Eevee.png');
```

(The image itself ('Eevee.png') can be downloaded from the course webpage.) There are two complications in using `imread`, however. To see what these complications are, after the `img_eevee =` line, add the following:

```
h_eevee = imshow(img_eevee,'Parent',h_axes);
```

to also display the image. Well, it is a bit of an odd mess. By default, `imshow` tries to fit an image to the space, so that might, at least partly, help explain things.

We can start by making the Pokemon image smaller and see whether that helps us to work out what is going on. To do this, we could e.g. pick half of the size of the **Axes** object, and plot the Pokemon from the origin. A replacement line to do this would look like:

```
h_eevee = imshow(img_eevee,'Parent',h_axes,'Xdata',[0 x_max/2],...
'Ydata',[0 y_max/2]);
```

When you run this, you should get Figure 5.9.

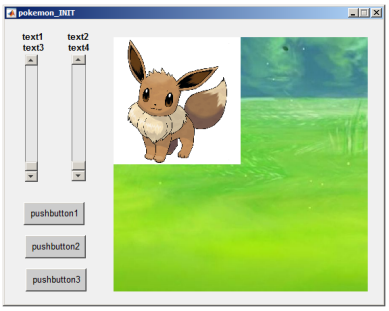

Figure 5.9: Template App with background image plus Pokemon.

You can see firstly that the Pokemon image is half the size of the space – exactly as we requested via `'Xdata',[0 x_max/2]` which says to start the image at zero on the x-axis and stretch it horizontally until half way along (`x_max/2`), and similarly for the y-axis. Except ... with `imshow`, it seems that the y-axis origin starts at the top and is positive downwards (which the Pokemon is in the top left, rather than bottom left, corner).

To cut a long story short, we can generalize the position and size of the Pokemon that is displayed (and use this at the end when we refine the App), via the following code fragment[28]:

[28] You should delete the lines starting `img_eevee =` and `h_eevee =` first. This 10-line code fragment then follows directly on from the previous 11-line one.

```
% define pokemon size
dx_pokemon = 0.2*x_max;
dy_pokemon = 0.2*y_max;
% define initial pokemon position
x_pokemon = x_max-dx_pokemon;
y_pokemon = y_max-dy_pokemon;
% read in pokemon image
img_eevee = imread('Eevee.png');
% draw pokemon
h_eevee = imshow(img_eevee,'Parent',h_axes,'Xdata',[x_pokemon...
x_pokemon+dx_pokemon],'Ydata',[y_pokemon y_pokemon+dy_pokemon]);
```

Now giving you a small Pokemon – in fact, 20% of the **Axes** size as specified in the definition of the Pokemon size parameters, `dx_pokemon` and `dy_pokemon`. If you run this, you should get Figure 5.10.

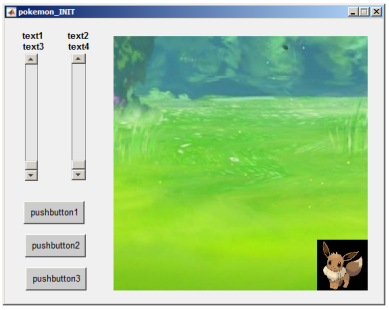

Figure 5.10: Template App with background image plus small Pokemon at bottom right.

One final thing now is the background to the Pokemon image. The original format (png) is actually defined with a transparent background. **MATLAB** can make use of this with a small tweak to the code – replacing the `img_eevee =` line with:

```
[img_eevee, h_map_eevee, h_alpha_eevee] = imread('Eevee.png');
```

which grabs additional graphics information and specifically about the transparency. And after the last line (`h_eevee =`), add:

```
set(h_eevee, 'AlphaData', h_alpha_eevee);
```

which implements the transparent background and hopefully gives you Figure 5.11.

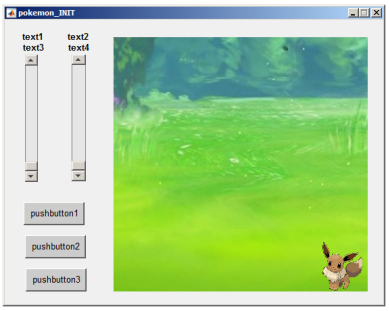

Figure 5.11: Template App with background image plus small Pokemon at bottom right, now with its transparency applied.

Part III – incorporating the ballistics model.

Here – almost all the code in this section will go into `function pushbutton1_Callback` – the function that is executed when the first **Push Button** is clicked. But before any coding – ensure that the text label associated with the first `Push Button` is appropriate for launching the ball ('Throw', 'Go!', whatever).[29]

Below is a simple rendition of the ballistics model. All that has been modified from a stand-alone **m-file** that would plot the trajectory of a ball, is that the creation of a figure (and associated `hold on`) is not necessary (because this has already bene done within the initialization function). either copy-paste your own version (and comment out the figure creation line), or add the below version.

```
% model constants
g = 9.81;
% model parameters
theta0 = 80.0;
s0 = 5.0;
h0 = 2.0;
```

[29] Remember – double-click on the **pushbutton1** object in the design editor and then find and edit the value of the **String** property.

```
% model parameters - time (s)
dt = 0.05;
t_max = 10.0;
% calculate initial velocity components
u = s0*cos(pi*theta0/180.0);
v = s0*sin(pi*theta0/180.0);
% set initial position of ball
x = 0.0;
y = h0;
% create Figure window and hold on
%Figure;
%hold on;
% run model
for t=dt:dt:t_max,
    % update horizontal and vertical positions
    dx = dt*u;
    x = x + dx;
    dy = dt*v;
    y = y + dy;
    % plot current position of ball
    scatter(x,y);
    if (y < 0.0)
        break;
    end
    % update vertical velocity (horizontal velocity unchanged)
    dv = -dt*g;
    v = v + dv;
end
```

When you rn the complete App, and press the first **Push Button**, you should see the balls' trajectory plotted. Upside-down! WTF!?

Well, this does seem to be the coordinate system in this **Axes** object. We can fix this by subtracting the model calculated height (`y`) from the maximum y-axis value (`y_max`) and adjust the `scatter` code line to:

```
scatter(x,y_max-y);
```

Except ... we defined `y_max` in the initialization function, and its value is not available in this function, unless we define it as `global` in both, so lets do that – add the following lines:

```
global x_max;
global y_max;
```

to both

- `function pokemon_OpeningFcn`
- `function pushbutton1_Callback`

(before any of your other code in these files, but below anything that **MATLAB** generated automatically in the first place).

It works, and in the right direction (for 'up'), but it is hardly **iTunes** grade App material. What we can do, is to replace the point plotted by `scatter`, with an image.

At the top of `function pushbutton1_Callback` (after the `global` declarations) load in a ball image:

```
[img_ball, h_map_ball, h_alpha_ball] = imread('Pokeball.png');
```

(using the full format of returned parameters because we'll make use of its transparency). We'll then define the size of the ball:

```
dx_ball = 0.05*x_max;
dy_ball = 0.05*y_max;
```

and finally, in place of `scatter ...`, write:

```
h_ball = imshow(img_ball,'Parent',h_axes,'Xdata',...
[x x+dx_ball],'Ydata',[y_max-y y_max-y+dy_ball]);
set(h_ball, 'AlphaData', h_alpha_ball);
```

The first of these final two lines, displays the image given by the parameter (ID) `img_ball`. It ensures it is displayed in the axes area pointed to by `h_axes` (and because of this, you also have to define `x_axes` as global[30], i.e. `global h_axes;`). Its size is `dx_ball` by `dy_ball`. Its x-coordinate is simply `x` (hence the image goes from `x` to `x+dx_ball`) and its y-axis coordinate ... well, don' worry about it, after much trial-and-error, it works. Now you should have something like Figure 5.12 when you run it.

[30] Directly underneath the other two `global` definition lines AND in a similar position in the initialization function: `function pushbutton1_Callback` .

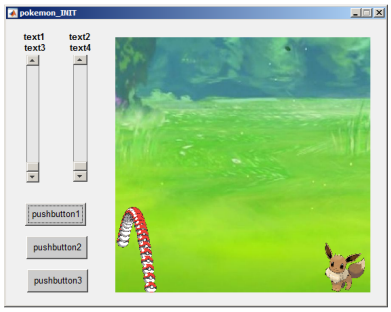

Figure 5.12: App with ball trajectory trail.

To finish this section off, we'll improve how the trajectory f the ball is displayed. Firstly, we could add a delay between each addition of the ball image, rather than them all sort of appear at once. After the `set ...` line, add:

```
pause(0.005);
```

This is some improvement visually. We could also remove the previous ball image, so that only one ball image is displayed on the screen at any one time, hopefully giving the impression of movement. Since we were good and obtained the handle (h_ball) of the ball image when we displayed it, this gives us a means to tell **MATLAB** to get rid of it again. Now, after the `pause` line, add:

```
delete(h_ball);
```

which simply deletes the last ball image object that was plotted.

Now when you run it you should see a single ball image that follows the trajectory that you calculated with your time-stepping ballistics model.

Part IV – utilizing the sliders.

So far it is not much of a game – the values of the parameters determining the initial speed and angle of the ball are set in the code.

You could always edit the code, save, and re-run to replay the game with a different throw, but ... really(?)

The **Sliders** are there to allow you to adjust the two key parameter values and the 'Throw' (/'Go') button can be re-clicked on to then re-run the game. The **Sliders** are set up such that when you move the slider, its value changes. In designing the GUI and creating the objects you have already set the min and max values of the **Sliders** to something reasonable. What remains is to obtain the value of each **Slider** and pass that to your ballistics model.

The first step is to read the new **Slider** value when the slider is moved. Taking the example of the first **Slider** ('**slider1**') which controls the initial speed of the ball – we first need to request the handle (ID) of this **Slider**. As before, we use the `findobj` function:

```
h = findobj('Tag','slider1');
```

which simply asks for the handle (passed to variable `h`) of the object whose '**Tag**' is '**slider1**'. You then[31] use the `get` function to get the '**value**' (one of the properties of the object):

[31] On the next line.

```
s0 = get(h,'Value');
```

where here the value is assigned to the variable `s0` (initial speed). These two lines of code go in `function slider1_Callback` just after the comment lines (there is actually no other code (automatically generated) in this function as it currently stands).

While we're here editing this function, what else might be helpful to happen when the slider is moved and its value changes? Although from creating the **Slider** object you know (unless you have forgotten) what the min and max **Slider** values are, you would still be somewhat guessing what its exact (or even rough) value was. During the GUI design phase, you created a pair of **Static text** boxes for each **Slider**. One of each pair was intended to display the **Slider** value. So lets do this now. The **Static text** box for the value display was called (its **Tag**) **text1**[32].

[32] At least, it was in my GUI design – check the name of yours.

Once again, before we can change any of the properties, we need to determine the handle of the object. For **Static text** box **text1**, the code would be:

```
h = findobj('Tag','text1');
```

(this should be starting to become familiar to you by now ...).

To set its value, which in this case is a text string, we write:

```
set(h,'String',num2str(s0));
```

where `num2str(s0)` converts a numeric value into a string (as you have seen before). These two lines of code will go after the first two

in the same function (as you need to have obtained the value of so before you can use it to change then text box display).

At this point you may as well save and re-run. Now, when you drag and release the slider for initial speed, its new value is displayed above it in the text box. At least, this should be what happens ...

Write the analogous four lines of code for the other Slider, which will go in `function slider2_Callback`. Now the parameter value being read and displayed in the text box is the initial angle of launch, `theta0` (of whatever you prefer to call the parameter).

Again – save and test what you have so far. This should now be two **Sliders** that are linked to two **Static text** boxes such that when the slider is moved, the new values are displayed.

There is one final step to take. If you change either or both **Slider** values and click on 'Throw' /'Go', the trajectory of the ball is the same as before – you are not actually changing the parameter values used to initialize the ballistics model yet. Recall that variables within *functions* are *private* – they cannot be 'seen' outside of the function their value is set in. Unless you declare them as `global` variables.

So, in each **Slider** function, you need to declare the respective parameter (`s0` or `theta0`) as `global`. This will need to be the first line of the code (after the comment lines and before the four lines of code you inserted). You will also need to add the global declarations at the start of the pushbutton1 code where your model lives (`function pushbutton1_Callback(hObject, eventdata, handles)`):

```
global s0;
global theta0;
```

You then need to comment out the lines that set your initial model parameter values:

```
%theta0 = 80.0;
%s0 = 5.0;
```

You can test it now, and if you do, you might find that nothing appears to happen if you press 'Throw'. Only if you change the slider positions does anything (i.e. a moving ball) happen. We have created the situation where the ballistics model takes it values for initial speed and angle from the parameters `s0` and `theta0`. The only place in the code in which these values are set are the **Slider** functions. BUT, the **Slider** functions are only called when the slider is moved. So on starting the App, unless you first move the **Sliders**, the values of `s0` and `theta0` are undefined[33].

What to do? Well, recall there is the function that is called when the App first starts up and in which we loaded up various images etc. In this function, we could also check the value of each **Slider**

[33] Invariably, undefined variables in code are assigned a value of zero, but you should never try and use a variable whose value has not somewhere been defined.

(even though the slider could not have been moved yet), set the parameter values, and display the **Slider** values in the **Static text** boxes.

At the end of the code in `function pokemon_OpeningFcn`, add:

```
% read in default model parameters and set labels
h = findobj('Tag','slider1');
s0 = get(h,'Value');
h = findobj('Tag','text1');
set(h,'String',[num2str(s0)]);
h = findobj('Tag','slider2');
theta0 = get(h,'Value');
h = findobj('Tag','text2');
set(h,'String',[num2str(theta0)]);
```

which is pretty well much just an amalgamation of the code you have added to the two **Slider** callback function. The last final piece is to remember that the initial Slider values you read and set `s0` and `theta0` on the basis of, cannot be seen outside of this function. So at the top, along with the other `global` statements, make `s0` and `theta0` global to.

Note that if you do not like the new defaults for `s0` and `theta0`, you can always edit the properties of the **Sliders** in the GUI design editor window thing.[34]

[34] Equally, you could have coded in defaults and then set the **Slider** values to be these defaults when the App starts up. The process is basically exactly the same as for setting the **Static text** box string values.

Part V – pokeball/Pokemon collision detection.

Part VI – final game refinements.

6
zero-D / equilibrium modelling

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

6.1 *A zero-D Energy-balance model of the climate system*

Box, or zero-D models need not involve the reservoir of a substance (e.g. trace metal, carbon, or nutrient concentrations) *per se* – the reservoir and fluxes of energy (heat) will do just fine. Which leads us to the climate system.

In this Section, you are going to create, and then use in a series of applications, a zero-D equilibrium global 'climate model' – the simplest representation of the energy-balance of the Earth's climate that it is possible to make. The model assumes that the climate system is in balance, with no net gain or loss of energy, and hence that the energy absorbed from incoming (short-wave) solar radiation equals the (long-wave) radiative loss from the Earth's surface (or top-of-the-atmosphere). The equations are outlined in the Box and you'll need to rearrange them in terms of T (mean global surface temperature).

The exercises that follow are structured and you need to pay attention to which **m-files** you are creating from scratch, which ones, having been created and coded up, you do not then further edit ...

8.1.1 In this first Subsection ('*The basic EBM*'), you'll create a script (# `scr_1`[1]) containing the Energy Balance Model (EBM), and test it.

(See Figure 6.1.)

8.1.2 Next, you'll turn your EBM script (`scr_1`) into a function (`fun_1`)[2] – passing in the solar constant and albedo as parameters, and returning the surface temperature. (And test it.)
(See Figure 6.2.)

8.1.3 In the Subsection '*Parameter sensitivity experiments using the EBM – #1*', you will create a new script (`scr_2`) with a single loop in it. Within the loop, you will make a call to the EBM function (# `fun_1`) that you created.[3]
(See Figure 6.3.)

8.1.3 Then, in an extension to the previous Subsection work, you will create another new script (`scr_3`), this time with a double (nested) loop in it. As before – within the loop, you will make a call to the EBM function. Note that there is going to something of a diversion in this Subsection that will illustrate nested loops for you.
(See Figure 6.6.)

8.1.4 In the penultimate Subsection ('Calculating the evolution of the solar constant'), you'll create a new function (`fun_2`), which will take time (counted from the formation of the Sun) in *Ga*, and return the value of the solar constant at that time ($S(t)$ (Wm^{-2})).
(See Figure 6.9.)

Energy balance modelling (1)

The surface energy budget at the Earths surface, to a zero-th order approximation, can be thought of as a simple balance between incoming, sort-wave radiation that is *absorbed*, and out-going, infra-red radiation.

On average (over the Earths surface and annually), the energy flux per unit area received from the sun, can be written:

$$F_{in} = \frac{\alpha \cdot S_0}{4}$$

(the $\frac{1}{4}$ appears because the cross-sectional area of the Earth is $\frac{1}{4}$ of its total surface area – i.e. you take energy intercepted by the Earth, which has an effective area of $\pi \cdot r^2$, and spread it out over the entire surface – an area of $4 \cdot \pi \cdot r^2$).

Albedo (α) varies hugely across surface types (and angle of incoming radiation). A commonly used mean global approximation is to set: $\alpha = 0.3$.

Net outgoing infrared radiation proceeds according to black body emissions:

$$F_{out} = \epsilon \cdot \sigma \cdot T^4$$

where ϵ is the emissivity, σ is the Stefan-Boltzmann constant (in units of Wm^{-2}), and T the temperature in Kelvin (K) ($273.15K == 0.0°C$).

For a perfect black body radiator, we would set ϵ=1.0. However, it turns out that the Earth is not a smooth and perfectly matt black sphere radiating directly from the surface to space ... there is an atmosphere and water surface over ~70% of its surface etc etc. A common modification is then to reduce the effective emissivity of the surface to less than 1.0. A value of 0.62 is given in *Henderson-Sellers* [2014], making the expression for the out-going flux:

$$F_{out} = 0.62 \cdot \sigma \cdot T^4$$

[1] This is not a suggested name of the **m-file**, but an ID to help you not get confused as to which script or function is being referred to in the text ...

[2] **Once the EBM function has been created, you do not at any point edit it any further!**

[3] DO NOT put code the loops into the EBM function – leave the function alone ...

And then ...

8.1.5 ... finally (Subsection '*Evolution* of Earth's surface temperature'), you'll create one last script (`scr_4`), with a loop in time in it, and from within this loop, you'll call first the solar constant function (`fun_2`), taking time as an input and returning the value of $S(t)$, which you will then pass into the EBM (# `fun_1`), returning T.
(See Figure 6.10.)

6.1.1 The basic EBM

To kick off – create a new script (**m-file**) ('`scr_1`' in the summary notation) and code up the analytical solution to the basic global mean energy budget at the surface of the Earth (see Box) in a program structure illustrated schematically in Figure 6.1.[4] The equations for in-coming and out-going radiation (energy) were given previously. You simply need to re-arrange these and write them as code. This will form the basis of subsequent, more complex (and later, time-stepping) models. You will need to find (from the Internet?) the values of the constants you need ... and will need to be careful with units of these.

For now – prescribe the value of S_0 – for which the modern value is 1368 Wm^{-2} as well as the value of surface albedo ($\alpha = 0.3$ by default) – somewhere near the start of the program. Then run it.

If you found a reasonable value for the solar constant, and did not screw-up the units on the Stefan-Boltzmann constant, then you should have an equilibrium (global, annual mean) surface temperature of around 14°C ... If not – debug. Assuming that the code ran without errors but gave a nutty answer:

1. Check that the units are correct.
2. Check that the equation has been re-arranged correctly – a common root of errors is incorrect placement of parentheses ... or not placing parentheses around multiple variables you are divining something all by.
3. If still 'no' – maybe take the 2 component equations (for F_{in} and F_{out}), plug S_0 into the equation for F_{in} and then play with different values of T to find a value for F_{out} that is approximately equal – is the value for T sane? If not, double-check the units and values in both component equations.
4. If still 'no' – WHAT HAVE YOU DONE?

Once it is working, have a quick play about, changing the value of S_0 and albedo (α) (saving the **m-file** each time and re-running) to get a vague feel for how sensitive the surface temperature is to these two parameters.

[4] Note that the code is relatively simple and does not involve (yet) loops or conditionals or anything like that. Although ... I am sure it will involve lots of nice juicy comments and sensible variable names(?)

Simply set up the values of the various constants and parameters you need at the start of the code, then solve for T at the end of the code. The structure (omitting `% comments`) of your code may look like:

```
% section for constants
(variables you do not expect
ever to change)
...
% section for parameters
(variables you might adjust)
...
% solve for T
T = ...
```

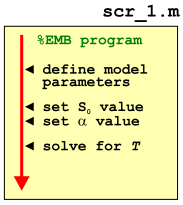

Figure 6.1: Form of the basic EBM model.

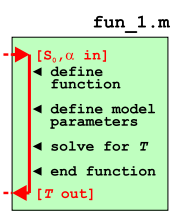

Figure 6.2: Form of the basic EBM model as a *function*.

6.1.2 *The EBM as a function*

We'll now make your model mode flexible so that it can be applied to the subsequent Examples. So – turn it into a *function*[5] that takes in 2 parameters – the solar constant (S_0) and the mean global albedo (α). The function should return the global mean surface temperature, T.[6] (See Figure 6.2)

Try playing with the function in the same way as before, but now passing the different values of S_0 and α (rather than having to edit the **m-file**, save, and re-run each time). To use the function (assuming you called it e.g. `fun_1`), and assuming the 2 passed parameters are in the order: S_0, α and are given their default values, you'd write (at the command line):

```
» fun_1(1368.0,0.3)
```

(and get a value close to 14°C returned).

6.1.3 *Parameter sensitivity experiments using the EBM – #1*

Now to utilize your new function ('`fun_1`' in the summary notation). Create a new blank script ('`scr_2`') and define 2 parameters near the start – one for the value of S_0 and one for α, then further down the code, call your function (`fun_1`), passing it these 2 parameters. So far so boring, as this is in effect what you had been doing in 'playing' with the function previously.

Common in numerical modelling is quantifying how sensitive a system is to the choice of parameter values – called a *sensitivity experiment*. You may already have gotten a feel for roughly how sensitive T was to changing S_0 on its own, or changing α on its own, but what about when both parameters vary together?

Lets start with a simple 1-D case, and consider just a change in the value of S_0. To automate generate different values of S_0 and call the function, you are going to need a loop[7]. There are two ways of constructing the loop[8]:

loop option #1 You could loop directly through the range of values of S_0 that you are interested in, e.g.

```
for S0 = 1000:100:1500
    % CODE GOES HERE
end
```

in which S_0 will go from 1000 to 1500 Wm^{-2} in steps of 100 Wm^{-2} [9].

Perhaps a little inconveniently, this does not pass through the modern value (1368 Wm^{-2}), although when you plot as a continuous line (e.g. in `plot`) or otherwise interpolate the results, maybe

[5] Refer to earlier in the text and also **help** on the required structure/syntax of a *function*. Recall the basic structure of a function **m-file**, has as its VERY FIRST LINE:

```
function [OUT] = ...
FUNCTION_NAME(IN)
```

where OUT represents one (or more) variables that are passed out (the 'result' of the function), `FUNCTION_NAME` is the name of your function, and IN is the name (or names, comma-separated) of one (or more) variables (parameter values) that are passed into the function. (The very last line of the function should have an `end`.)

For example, to pass in two variables, `IN_1` and `IN_2`, you'd have:

```
function [OUT] = ...
FUNCTION_NAME(IN_1,IN_2)
```

[6] Note that the parameters passed into, and returned by, the function, can be called anything you want. As long as they are useful (and clearly defined/explained in a comment somewhere).

[7] You are going to put the loop in the function (# `fun_1`), NOT the script (# `scr_2`).

An entire plane of Hell is reserved for anyone coding the loop in the function.

[8] In both cases a `for ...` loop.

[9] You can pick a different range and increment ... this is just a quasi-random example to illustrate ...

this does not matter. You could have addressed this by constructing a slightly less convenient form of the loop, e.g.:

```
for S0 = 1068:100:1568
    % CODE GOES HERE
end
```

which now passes exactly through the modern value of S_0.

loop option #2 Alternatively, you could have an integer count for the loop, and then derive a changing value of S_0 from this. For example:

```
S0_modern = 1368.0;
for m=-5:5
    S0 = S0_modern + 100*m
    % CODE GOES HERE
end
```

Look carefully through this code and follow what is going – as `m` counts from `-5` to `5` (in steps of `1`), `100` times the value of `m` is added to the modern value of S_0[10], meaning that S_0 ends up going from `S0_modern` - 500, to `S0_modern` + 500 Wm^{-2} (in steps of 100 Wm^{-2}).

[10] The variable definition `S0_modern = 1368.0` at the top of the code fragment.

Or, alternatively:

```
S0_modern = 1368.0;
for m=1:11
    S0 = S0_modern + 100*(n - 6)
    % CODE GOES HERE
end
```

which does exactly the same (do a mental check on this) but now counts `m` starting from a value of `1`.

So what does it matter, and/or is one 'better' than the other? Actually, both are equivalent and you could make either work out just fine. The advantage with the second version is that you implicitly have an integer counter. For the first version, you'd have to add lines, e.g.:

```
count = 0;
for S0 = 1068:100:1568
    count = count + 1;
    % CODE GOES HERE
end
```

And why might we want some sort of an integer counter in the first place? Well, you might want to save the data(!), i.e. the calculated (by your function) value of T vs. the inputted value of S_0.

There are also two ways of saving the data (assigning calculated values to sequential locations in an array):

save option #1 Create the necessary array(s) beforehand, e.g. using the `zeros` *function*. For instance, to create a vector with 11 rows (and 1 column), suitable for saving the value of T calculated by each call to the EBM function, you could write:

```
data_T = zeros(11,1);
```

which would create a (single) column vector with 11 rows. You'd need an equivalent vector (e.g. `data_S0` in this example) for storing the corresponding value of S_0 used in the temperature calculation. These vectors are created before the loop starts.
Then within the loop (and after the calculation of T), you'd assign your values of S_0 and T by using whichever index you created[11]:

```
data_S0(m)  = S_0;
data_T(m) = T;
```

or:

```
data_S0(count)  = S_0;
data_T(count) = T;
```

where `m` and `count` are integers, starting at a value of one, and incrementing by a value of one on each successive execution of the loop. `m` (or `count`) represents an index that allows you to store the result of each successive calculation (as well as the corresponding input value) in a vector.

save option #2 Or ... **MATLAB** will allow you to 'grow' a vector, one element at a time (but not for matrices).[12] The the code within the loop actually looks identical – you just omit the 2 lines at the start of the program creating vectors of appropriate size (and zero in value).

So pick one (i.e. a way of saving a pair of values each time around the loop) and code it up. (Or try both!) Then, at the end of your program, plot (`plot` or `scatter`) how T varies as a function of S_0.

The structure of your code should look like Figure 6.3. and your resulting figure (depending on the range you assume for S_0), something like Figure 6.4.

[11] i.e. which of the two OPTIONS you chose earlier.

[12] The vector automatically grows in length as you add values to it. If you don't believe me, try the following:

```
» A=1;
» A(2) = 2;
» A(3) = 3;
```

You could instead define at the start f the code (before the loop) a vector of zeros of the correct length, the 'correct length' being the number of time around the loop. See function `zeros`. Or even `NaNs` ...

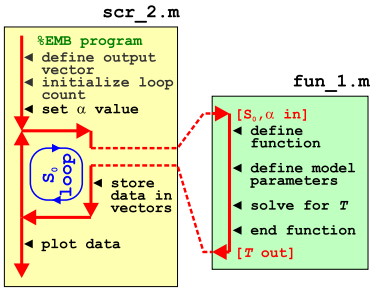

Figure 6.3: Schematic structure of the model configured to carry out a single parameter sensitivity study.

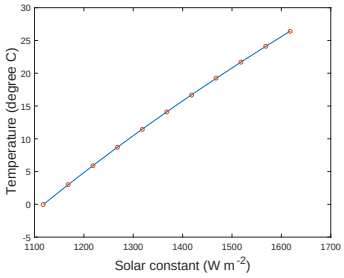

Figure 6.4: Sensitivity of global mean surface temperature vs. solar constant (mean surface albedo held constant at an albedo value of 0.3).

6.1.4 *Parameter sensitivity experiments using the EBM – #2*

In this Subsection, we'll extend the sensitivity experiment to 2D, assuming that you are interested in how T also varies as a function of α. So, you'll need to vary both S_0 and α, and in all combinations of the two. In fact, in a grid pattern, with S_0 increasing in steps on one axis (as before), and α on the other.

Hopefully, you might have guessed that you'll need a *nested loop*(?) – one loop going through all possible values of α, *for each and every possible value of* S_0??

Perhaps, as an aside, we'll go through a simpler example/system first.

A chess board consists of squares in a 8×8 grid. The squares alternate black and white. To define 8 squares (points) along the x-axis on the bottom row, you'd write something of the form:

```
for m=1:8
    % SOME CODE GOES HERE
end
```

Now, if you wanted to define 8 squares along each column (the y-axis), at each and every x-axis value, you'd need to loop through all the rows, So you need a loop in e.g. `n`, inside the loop for `m`:

```
for m=1:8
    for n=1:8
        % SOME CODE GOES HERE
    end
end
```

Follow this through to satisfy yourself that for each and every value of `m` from `1` to `8`, `n` loops from `1` to `8`, and hence visits every point in turn of a 8×8 (`n,m`) grid.

Actually, now we have got this far, it is good practice to consider how we'd define the black and white squares. We'll assume that black is represented by `'1'` (*true*) and white by `'0'` (*false*) and create a board (array) of all white squares to start with, i.e.

```
board = zeros(8);
```

(Refer to **help** or earlier for the syntax for help on the function `zeros`.[13])

If we start with a black square (`'1'`) at the bottom left, we could define an *algorithm* for creating the grid as: odd column number squares are black, as long as the row number is odd, otherwise they are white.[14] So to implement this in code – as we loop through both column (`m`) and row (`n`) on the board, we test for the column number being odd and row number odd, OR, the column number being even and row number being even. If *true*, the square is defined as black. The only tricky bit is to determine whether the row or column number is even or odd. We do this by testing whether there is any remainder after dividing by `2`, using the function `mod`.

The complete code looks like:

```
board = zeros(8);
for m=1:8
    for n=1:8
        if ((mod(m,2)>0 && mod(n,2)>0) || (mod(m,2)==0 && mod(n,2)==0))
            board(n,m) = 1;
        end
    end
end
```

[13] You could alternatively write this:

```
board = zeros(8,8);
```

`mod`

Not ... the opposite of `rocker` (which doesn't exist in **MATLAB** anyway) but short for *modulo*. Wikipedia helpfully tells us:

"*In computing, the modulo operation finds the remainder after division of one number by another (sometimes called modulus).*"

Or in MATLAB-speak:

```
b = mod(a,m)
```

"*returns the remainder after division of* `a` *by* `m`*, where* `a` *is the dividend and* `m` *is the divisor*".

It turns out that as long as a is positive, you can use to test for whether an integer `a` is *even* or *odd* by:

```
b = mod(a,2)
```

When the returned value `b` is `0`, `a` is *even*, and when `b` is `1`, `a` is *odd*.

[14] Look up a picture of a chess board to convince yourself that this works.

Spend a little time decoding the `if` statement for practice ... If you want to see that it works – code it in a new **m-file**, run it, and then plot up `board` by e.g. using `imagesc` (cf. Figure 6.5). Beautiful.

OK – that was easily the greatest diversion in pedagogical history, but nested loops should now come almost as second nature to you :o) So how about coding up the nested loop for the question we were meant to be addressing – carrying out a 2D sensitivity test of the parameters S_0 and α. See if you can create this.

Start with a new (script) **m-file** ('`scr_3`'). For constructing the loop – you have already seen the 1D example of parameter sensitivity code, and also an example of creating a nested loop for a 2D grid. Your chess board columns (`m`) become S_0, and rows (`n`) become α. You don't need to do anything so awful as that `if ...` statement – instead just call your function (`fun_1`) for solving the global surface temperature (passing it the values of S_0 and α generated in the loop). A schematic of the program structure is shown in Figure 6.6.

For saving the data (within the loop), you cannot not simply index the locations you want in a 2D array (matrix) that did not previously exist and expect it to 'grow' as before, becasue a matrix must have all complete rows and columns. Instead, near the start of the code (before the loop), create a matrix of the size of the parameter grid. For example, if you were going to loop through 10 different values of S_0 and 10 of α, you could write:

```
data_output = zeros(10);
```

(creating a 10×10 array of zeros). Or if for example, you had 20 different values of S_0, and 10 of α:

```
data_output = zeros(10,20);
```

(20 columns times 10 rows).

Within an (`n`,`m`) loop you then assign your calculated value of T to the appropriate location:

```
data_output(n,m) = T;
```

Don't forget that you'll also need to know the values of S_0 and α that correspond to the column and row numbers. Perhaps save these as 2 individual vector (as per before) or ignore them for now.

One slight complication if you use a pair of counters and increment their value each time around their respective loops (rather than having a integer count for the loop itself (i.e. `n` and `m`)) – the innermost counter must be reset in value each time the outer loops starts:

```
count_outer = 0;
for ...
    count_outer = count_outer + 1;
```

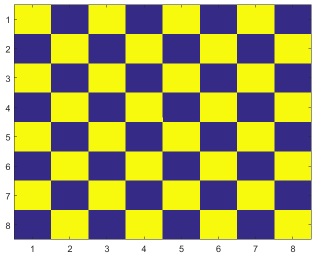

Figure 6.5: Chess board grid pattern.

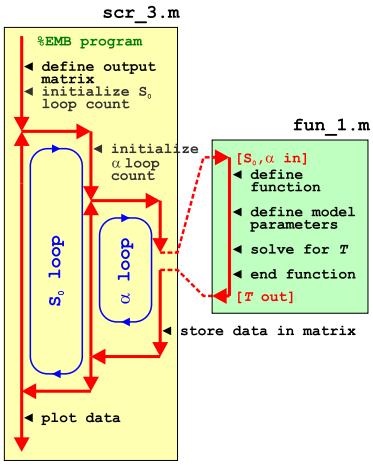

Figure 6.6: Schematic structure of the model configured to carry out a double (in terms of solar constant AND now albedo) parameter sensitivity study.

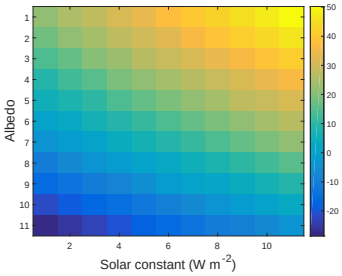

Figure 6.7: Global mean surface temperature (°C) as a function of solar constant and surface albedo grid point number.

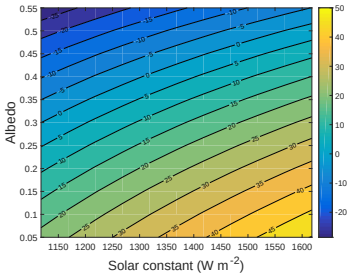

Figure 6.8: Global mean surface temperature (°C) as a function of the value of solar constant and surface albedo.

```
    count_inner = 0;
    for ...
        count_inner = count_inner + 1;
        % CODE GOES HERE
    end
end
```

(Try it instead by initializing both prior to the outer loop, and see what happens ...)

When you *think* you have this working and generating a matrix of T values[15], plot the resulting surface of T vs. the two parameters. Rather than using e.g. `imagesc` (Figure 6.8)[16], try `contour`[17] or `contourf` (e.g. Figure 6.7).

[15] HINT: create a 2D array of the appropriate size first, before the *loop* starts, using `zeros`, and then populate it with the values of T as the *loop* loops.

[16] Note that the temperature grid points are plotted as a function of column and row number and that the plots ends up 'up-side-down' compared to the `coutourf` version.

[17] You'll need to employ `meshgrid` based on the same 2 vectors of values that the loop creates for S_0 and α.

6.1.5 *Creating a function for the evolution of solar constant through geological time*

In this and the final Subsection, you are going to leave the 2D-ness aside and consider how Earths surface temperature has changed through geological time.

So far you only have a function equating solar constant (S) to temperature (T). What you need is some way of equating time (t) to the value of the solar constant at that time S_t (which you can then turn into temperature). We'll remedy this toot sweet.

Start by creating a new (blank) **m-file** and define it as a *function* that takes in time (in units of *Ga*) and spits out S_0 (Wm^{-2}) (this will be '`fun_2`' in the on-going notation).

The background to the equation that will go into your function is given in the **Solar constant Box**. In this, you'll first need to substitute the modern value of the solar constant into the equation to leave it in terms of S_t (the solar constant value at time t) rather than L_t. Your function, aside from the all-important 1st line (and `end` at the end) and appropriate `% comments`, need have little more in than a definition for any constant you might want to use, such as the modern value of S_0 and perhaps time now (4.57 Ga) ... and a single line for the equation giving the value of S_t. Be careful that in the equation, t is measured as the age of the Sun (since its formation), meaning that time 'now' (modern), is equivalent in the equation to $t = -4.57$ (Ga).

When you think you have done this – check it – plug in values of time into your function, i.e.

```
» fun_S(4.57)
```

for passing the time now into a *function* called '`fun_2`' in the on-going notation (which in this example should return a value of 1368 (Wm^{-2})).

Solar constant

The long-term evolution of solar luminosity L_t as a function of time t can be approximated [*Gough* [1981]; *Feulner* [2012]) by:

$$\frac{L_t}{L_0} = \frac{1}{1+\frac{2}{5}\cdot(1-\frac{t}{t_0})}$$

where t_0 is the age of the sun – 4.57 Gyr (4.57×10^9 yr) and L_0 is the present-day solar luminosity (3.85×10^{26} W).

The value of L_0 is equivalent to a flux (Wm^{-2}) of 1368 Wm^{-2} incident at the top of the atmosphere at Earth, which is given the symbol S_0. In the equation, L_0 can be substituted for S_0 to give the value of S at ant time, i.e. S_t (Wm^{-2}).

Note that in the formula, t is counted (in Gyr) relative to the formation of the Sun (i.e. present-day would be: $t = 4.57$).

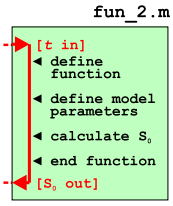

Figure 6.9: Schematic structure of code for calculating the solar constant (output) as a function of time (input).

6.1.6 *Using multiple functions and calculating global surface temperature as a function of geological time*

Finally ... you are going to bring it all together and calculate and plot the surface temperature of the Earth, at 100 Myr intervals, from 4.0 Gyr (4 billion years) in the past, to 4.0 Gry in the future – spanning approximately the age of the Earth and much of its potential long-term future.

Start by creating one final new (blank **m-file**) script ('`scr_4`').. You are going to need a loop in time, perhaps looping from 4.0 to -4.0 Ga relative to now (but you can chose what limits you like ... except remembering the Sun is only 4.57 Ga old ...). Within the loop, you will:

1. Pass to your solar constant function the current time, and obtain the corresponding value of the S_t – remember that you must add 4.57 to the time you pass into your function as the equation for S_t is in terms of time since the formation of the Sun, not relative to now.
2. Call your EBM function to calculate the corresponding surface temperature, passing it the value of S_t you have just calculated.
3. Store in an array, or pairs of vectors, time and the corresponding value of T.

Likely bug possibilities include the units of time (Gyr), and that time in the equation for S_0 is counted forwards from the formation of the Sun. Also be careful with nested parentheses (()). A schematic of the program structure is shown in Figure 6.10.

Assuming that you have managed something like Figure 6.11[18] – what strikes you, in light of (hopefully) what you know about the past history of climate and evolution of life on this planet, about your model projection (for the past)? What is 'missing'?

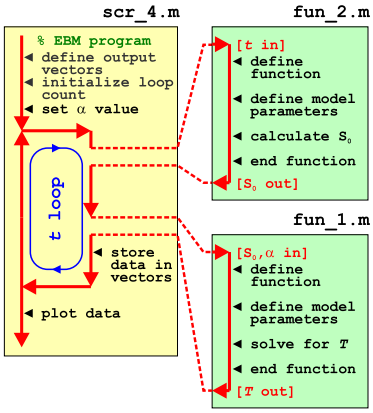

Figure 6.10: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, and solar constant and EBM functions.

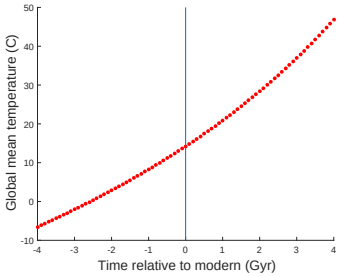

Figure 6.11: Simple EBM projection of the evolution of Earth surface temperature with time. Time at the present-day is highlighted by a vertical line (drawn using the **MATLAB** `line` function).

[18] Note that a line has been added to highlight $t = 0$ (i.e. the present-day) – see `line`.

`line` ... quite simply, draws a line. The basic syntax of the command is:

```
line(X,Y)
```

which plots a line between a paid of (x,y) coordinates. In the MATLAB usage, for a single straight line segment: the vector `X` contains both the x coordinate values, and `Y` both the y coordinate values.

In the specific Example in the text, the vertical line is drawn by:

```
line([0 0],[-10 50]);
```

NOT forgetting to put `hold on` first ...

6.2 *'Daisy World'*

There is an absolutely classic paper from the early 1980s – *Watson and Lovelock* [1983] – that illustrates how simple (biological) feedback on climate can lead to a close regulation of global climate over an appreciable span of the Earths past (and future). The premise for this model is a planet covered in bare soil (essentially, as per in the earlier EBM), but on which 2 different species of daisies (could be any pair of plants with contrasting properties) can grow – one white (high albedo) and one black (low albedo)[19]. Because the two species modify their local (temperature) environment and their net growth depends on how close the local temperature is to their optimum growth temperature, a powerful climate feedback operates and as the solar constant increases, the abundance of daisies switches from black to white – driving an increasing cooling tendency of the planet surface in the face of increasing solar-driven warming. This regulation emerges as a property of the dynamics of the population ecology and interaction with climate and does not require an explicit regulation of climate to be specified. Just dumb daisies doing their day-to-day stuff.

We'll code up this model ... but as before, in discrete stages (aka, the following Subsections).

[19] As pointed out in *Watson and Lovelock* [1983], the actual 'colors' are immaterial – just tat the albedos differ.

8.2.1 This will be the simplest addition to your previous model[20]. You'll create a new 'fixed daisy' function (`fun_3`) which will take no(!) inputs, and return a value for mean global albedo. You'll also copy-rename yourself a new script ('`scr_5`' – based on `scr_4`) and in it, take the albedo value generated by the call to the daisy function, and pass it into you EBM function (`fun_1`). (See Figure 6.12.)

[20] i.e. the one comprising a loop through time, and within this loop, calls to your function to convert time to solar constant, and take the solar constant (and albedoo and solve for mean global surface temperature. This was '# `scr_4`' in the previous Section notation.

8.2.2 Now, in the next stage it gets a little more complicated, because in a further new function ('`fun_4`' – copy-renames-and-edited from `fun_3`) you'll modify the equations such that the relative abundance of each daisy type is now responsive to the value of global temperature. The situation thus becomes – the relatively fractions of dark and light colored daisies is a function of global surface temperature, yet ... global surface temperature, through the mean (fractional area weighted) albedo of the daisies, is a function of the relatively fractions of dark and light colored daisies – a circularity (feedback loop). We'll resolve this circularity (i.e. come to a steady state solution) by creating an inner loop that comprises only the daisy function and EBM function and keeps looping until ... well, we'll start by simply prescribing a fixed number of iterations of the loop.

(See Figure **??** for a schematic of the code setup.)

8.2.3 Finally (almost) – we'll allow the daisies affect their *local * (temperature) environment. Now it gets more interesting (honest!). Although the code structure is exactly the same as in the last step[21], you will require a further copy-rename-and-edit of the previous daisy function ('`fun_4`' → '`fun_5`') and one further copy-rename-and-edit of the previous script ('`scr_6`' → '`scr_7`') that calls the daisy function.

8.2.4 In a minor extension to the previous work, we can modify the loop involving the daisy function and EBM function such that it will proceed until an adequately accurate solution (for global temperature) has been converged upon (rather than looping a fixed number of times).

6.2.1 *'fixed daisy' daisy-world*

To start: read *Watson and Lovelock* [1983]. You should be able to take away from this some of the essential information that you need to specify and keep track of. For now, we'll just concern ourselves with defining the albedo of bare ground (soil) and the albedo of each daisy together with how much area is covered by each species of daisy.

Create a new function (`fun_3`) – configure it so that it returns a single parameter – albedo. For now it has no inputs.[22] How it relates to your previous program and code for how the Earth's surface temperature evolves over geological time, is illustrated in Figure 6.12.

Now, in the daisy function (`fun_3`) near the top, define yourself some parameters for the daisy model:

```
% define model parameters - daisy albedo
par_a_s = 0.3; % albedo - bare soil
par_a_w = 0.5; % albedo - white daisies
par_a_b = 0.1; % albedo - black daisies
% define model parameters - daisy land fraction
par_f_w = 0.01; % (land) fraction - white daisies
par_f_b = 0.01; % (land) fraction - black daisies
```

(or using whatever parameter names you prefer). Here, the albedo values associated with each daisy type are fixed and will be used regardless of what the model does. The values have been chosen, assuming equal proportions of black and white daisies, to given an average of 0.3 – the albedo of bare soil and also the assumed value in the previous EBM. You'll modify and play with this value all too soon enough. The surface area fraction values are just initial values to start the model off with.[23]

Next, and actually the only line of any note in the function – you need to calculate an average albedo[24] – calculated based on the area

[21] A loop through geological time, as per in the previous Section. Within this main loop, you'll have a sub-loop with just the daisy function followed by the EBM function.

[22] A funny sort of function, although pretty well much like `pi`.

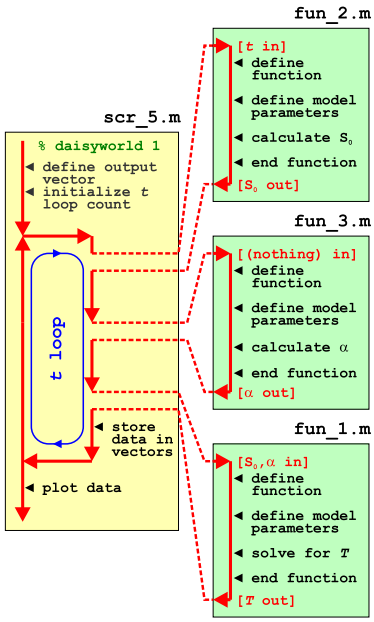

Figure 6.12: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, the solar constant and EBM functions, and now the 'daisy' albedo function.

[23] As you'll come to see subsequently, these cannot be zero. Or rather, a daisy species can start with a fractional area of zero, but you'll never ever get any of that species growing, regardless of the environmental conditions (because there are none to start with!).

[24] Note that it is very easy to accidently prescribe a total area covered by daisies of >100%. You should ideally put a check (`if ... end`) in the code before it tries to calculate anything for whether the total area initially covered by daisies exceeds what is possible. If this is the case, your code might spit out a warning message (a simple `disp` command would do). You might also terminate your program (see `exit`).

weighted average of: bare soil, white daisies, black daisies. The calculation is simple and you already have the areas of the two species of daisy as fractions. You weight the contribution to global albedo by the albedo of each daisy by its fractional area. You then just need to calculate the fraction of the Earths surface that is bare soil – the area fraction not covered by daisies. In maths-speak, the mean albedo is given be:

$$\alpha = F_w \cdot \alpha_w + F_b \cdot \alpha_b + (1.0 - F_w - F_b) \cdot \alpha_s$$

where α_w, α_b, and α_s, are the albedos of white and black daisies, and bare soil, respectively, and F_w and F_b are the fractional areas of occupied by white and black daisies, respectively (with bare soil comprising the remainder). You simply need to translate this into **MATLAB** code using the parameters you defined earlier (for α_w, α_b, and α_s, and F_w and F_b). Write this line of code, which the one and only calculation the function carries out, just before the `end` of the function.

Thats actually it. All the parameter values are specified and fixed (see above), so nothing particularly exciting is going to happen ... Regardless – run the the complete model with the value of albedo now depending on the fraction of white and black daisies – it should look identical to before in terms of the evolution of surface temperature with time (it must, because the default parameters above ensure that the mean albedo is always 0.3 and the daisies don't even know anything about growing (or dying) yet). Model (surface temperature) output, including how the populations of the 2 species of daisy also vary with time, is shown in Figure 6.13).

You might play briefly with the prescribed daisy fractions and albedo values and e.g. check that when you specify a configuration with 100% of land area covered by black daisies, the climate is much warmer throughout the simulation, and when white daisies are assigned an initial value of 1.0, the climate is always much cooler compared to in the default simulation.

6.2.2 *'dumb daisy' daisy-world*

OK – step #2 in the evolution of Daisy World, and for the next modification and one which will actually make something 'happen' (i.e. the simulation will be different to that of the default EBM based simulation of mean global temperature response to increasing S_0). In fact, the daisies are going to grow and die (but unlike Southern California, not burn), with their population changing over time until an equilibrium is reached (for a particular specified value of S_0). *Watson and Lovelock* [1983] give a simple population model formulation for

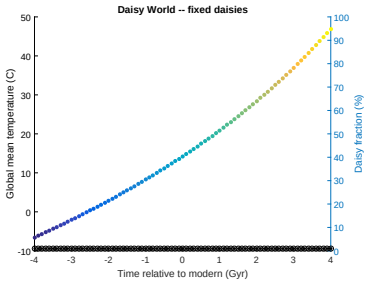

Figure 6.13: Evolution of global surface temperature and the two populations of daisies with time ... but with no change allowed in the daisy populations (d'uh!). The fractional coverage of white daisies is shown by large empty circles, and for black, by small filled black circles. Data points for mean surface temperature are color-coded by temperature (color scale not shown).

Daisy population dynamics (1)

For an area fraction occupied by white and black daisies of F_w and F_b, respectively, the change in occupied fractional area with time (t) can be written:

$$dF_w/dt = F_w \cdot (x \cdot \beta_w - \gamma)$$
$$dF_b/dt = F_b \cdot (x \cdot \beta_b - \gamma)$$

where x is the free (i.e. not occupied by daisies of any color) area of (fertile) ground, equal to:

$$x = 1.0 - F_w - F_b$$

(assuming here, unlike the more general case in *Watson and Lovelock* [1983], that all the land area is potentially fertile), β is a temperature-dependent growth function (one for each species of daisy), and γ the mortality rate (as a proportion of the area covered by that species of daisy per unit time). The value of γ given in *Watson and Lovelock* [1983] is 0.3, but this could be a parameter that you could play about with and investigate its effects.

To simplify things to start with, growth is a function only of the global mean temperature (in °C):

$$\beta_w = 1.0 - 0.003265 \cdot (22.5 - T)^2$$
$$\beta_b = 1.0 - 0.003265 \cdot (22.5 - T)^2$$

(where the value of 22.5 °C is a reference temperature and represents where optimal (maximum) growth occurs).

the change in area fraction covered by both sorts of daisy with time (also see Box) that we will implement here.

The unit of population in Daisy World is fractional area covered. So each time-step, the fractional area or each species will grow or shrink, depending on whether mortality is higher than growth. Both growth and mortality are formulated as being dependent on the fractional area (at the previous time-step), i.e. growth in covered area depends on how much is already covered. Similarly, mortality also depends on how many daisies are currently there. The growth rate is further modified by the available fractional area, such as that the area left shrinks, the growth rate shrinks. (Effectively, this is perhaps trying to account perhaps for shrinking resources available for further growth. It also has the effect of adding numerical stability to the model and helps presents over-shoots where the total fractional area covered by daisies far exceeds 1.0 ...).

How them to implement this in code?

- In general – start by identifying any constants – i.e. fixed and invariant, fundamental values, such as π or the Stefan-boltzmann constant. These values could be hard-coded into the equation as numbers, but better is to replace them with variables that you'd define at the top of the m-file as this makes for neater and easier-to read **MATLAB** code.
- Next identify any parameters – values that are not fundamental properties of the universe, but may be considered invariant for sequential uses of the equation. The characteristic albedos of the two species of daisies is a good example – these values are 'fixed', although, one day you might change them. If the code file is a script – define MATLAB variables and assign values to them, near the start of the code file. Otherwise, if a function, you may need to pass these parameters into the function and so they need to appear in the function definition on the 1st line of the code.
- Identify any output variables, i.e. result(s) of the calculation. In a function, these are invariably pass back out and hence need to appear in the function definition on the 1st line of the code. Output variable may also be input variables – i.e. a calculation may take the current value of a variable (as an input), update it, and then pass it back out. In which case, the variable will need ot appear as both input and output. Perhaps pick distinction variable names to avoid confusion, e.g. `var_in` and `var_out`.
- You may have local variables (i.e. used only within the script and out outside of it). If scalars, these need not be defined and initialized, unless used as e.g. a counting or running-sum variable. If in doubt, maybe also define and initialize e.g. to zero local variables.

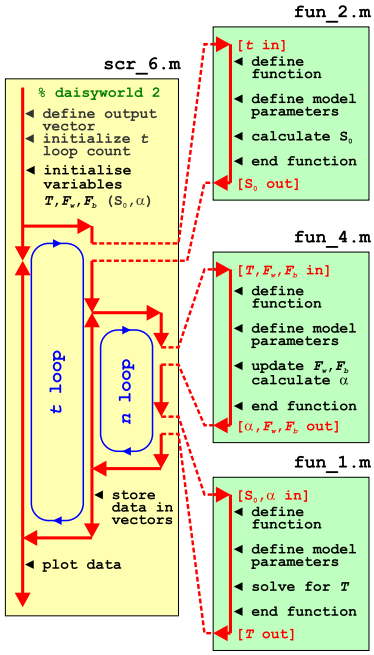

Figure 6.14: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, the solar constant, EBM, and 'daisy' albedo functions. Note the creation of an inner loop, with EBM, and 'daisy' albedo functions called from within this, while the solar constant remains called form the start of the outer loop as before.

- Otherwise, it is mostly just a case of writing the maths, in MATLAB – changing symbols where necessary and replacing the letters (invariably) used in the equations with your variable names.

Figure 6.14 gives a schematic of the overall code structure for this model. **DON'T PANIC.** There are actually only 2 (or 3-ish), relatively incremental changes, compared to previously. Start off by noting what is the same – both the function for the solar constant (`fun_2`) and the EBM model (`fun_1`) are exactly the same as before. The loop in (geologic time) and hence some of the script (`scr_6`) is also the same. What is different and yet to-do?

1. Lets start with the daisy function. You could deal with the inputs and outputs first. As as well as T, now the previous values of the fractional areas of the two daisies are required (F_w, F_b) (which is different from before where the values were assumed and the respective parameters set at the start of the function[25]). This is because each time the daisy function is called, the fractional areas are updated (hence why they are inputs). And outputs. Beasue the daisy function is updating the fraction al areas, these two parameters also need to be outputs too. So the very first thing to do is to modify the function definition, so that the inputs are:

   T, F_w, F_b

   and the outputs are:

   α, F_w, F_b

   (see help of various sorts on *functions*, but it not at all a fundamental change as to compared to before).

   Then, the only other development in the function, is to implement the equations for daisy growth/death and update the values of F_w, F_b (and at the end, calculate the value of α as before). And ... set the parameter values for β and γ of the two daisy species (near the start of the function).

2. Secondly, it is going to take a number of iterations for the daisies to grow/die ... changing their fractional areas and hence albedo as their fractional areas change ... and hence ultimately, reaching a new equilibrium with global climate. Each time around the outer loop – because the value of S_0 will change each time, climate will change and the daisy population will no longer be in equilibrium (because their fractional areas are carried over from the previous loop iteration). Hence in the outer loop you will need an inner loop to determine the new equilibrium and global temperature for that particular value of S_0. For now the loop can be

[25] So if you are copy-pasting the previous Daisy function, you need to delete the lines:

```
par_f_w = 0.01;
par_f_b = 0.01;
```

quite simple – we'll assume 100 iterations (i.e. the loop counter `n`, will go from `1` to `100`).

3. Lastly, the initialization of the main program (`scr_6`) will be a little different from before. Because the daisy function now takes as input, F_w and F_b – you'll need to give these variables each an initial value (near the start of the program) so that first time the function is called, there is a value for the equations to work with. Similarly, temperature T now also becomes an input to the daisy function (and it is not set anywhere else beforehand in the very first iteration of the loops), so it also needs an initial values to be assigned.[26]

If you have set this daisy population dynamics enabled EBM (a DPDE-EBM!) up correctly, and drive it with your -4.0 to +4.0 Ga solar constant calculating script, you should get something like Figure 6.15.

OK, so actually, this is not different in terms of the global mean temperature response (to solar evolution), to before. But then again, you have set both species of daisy with the same temperature growth response. In other words, as the white daisies with a high albedo grow, so to the black ones with a low albedo. Equally. And their different albedos balance, meaning that α still never changes. One thing you could try to liven things up a little is to change on of the value of β (and/or γ) so that their population dynamics are not identical. Now, if the relative abundance of white and black daisies changes, so too with global mean albedo and hence global temperature.

[26] For completeness, you could also initialize S_0 and α, but it is not strictly needed, as they are calculated and defined before they are first used.

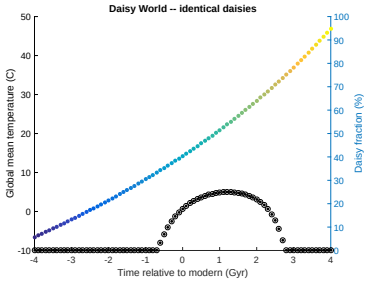

Figure 6.15: Evolution of global surface temperature and the two populations of daisies with time ... but now assuming that the growth of each depends on the global mean surface temperature.

6.2.3 *'clever daisy' daisy-world*

The last step is to give each species of daisy a different environmental preference for growth (why? because that is how the World works – different plants and ecosystems tend to inhabit different environmental regimes as a result of being (evolutionary) adapted to different environmental parameters). *Watson and Lovelock* [1983] assume that both species of daisy have the same temperature preference but modify their local environment differently – white daisies inducing a local cooling relative to the global mean temperature, and the presence of black daisies driving a local heating (see Box). The result is Figure 6.16.

Now the behaviour of the system and the evolution of global mean surface temperature with time, is very different. Towards the start of the experiment, and at very low values of S_0, the global mean temperature is too cold to support a daisy population (of either type). As the value of S_0 increases, initially global mean temperature follows the path it did before, in the absence of daisies (or with fixed, or equal populations). At a certain point, black daisies, because of their

Daisy population dynamics (2)

To make the different species of daisies interact differently with the environment, the temperature-dependent modifiers of growth are made functions of the local (to the daisy population or individual), rather than global, temperature:

$$\beta_w = 1.0 - 0.003265 \cdot (22.5 - T_w)^2$$
$$\beta_b = 1.0 - 0.003265 \cdot (22.5 - T_b)^2$$

There are all sorts of says of defining how the local temperature deviates form the global mean. In *Watson and Lovelock* [1983] this is simply reduced to a simple deviation that scales linearly with the difference between mean global and local (daisy) albedo:

$$T_w = T + q \cdot (A - A_w)$$
$$T_b = T + q \cdot (A - A_b)$$

(noting that A is albedo here, not alpha as was the case in the original (non daisy enabled) EBM). q is a simple scaling factor that describes how strongly the local temperature deviates from the mean (or conversely, how efficiently heat energy is mixed between differen daisy fractions) and is assigned a default value of 10.0.

advantage that they absorb more sunlight and drive a locally warmed climate, take off in population and rise to dominate 70% of the land surface. The global mean temperature transitions sharply to a much higher temperature state. As S_0 further increases in value, they increase slightly further in dominance (and global temperature climb a little further in response) until locally they reach their optimal temperature for growth. Past this (optimal temperature) point, white daisies start to grow and slowly replace the black ones. Global climate is almost perfectly stabilized during this interval. Beyond this, there is a short interval where black daisies die out and white daisies go on to reach their own (local) temperature optimum. Beyond this again, everything suddenly goes extinct in a rapid warming feedback of increasing temperatures, declining white daisy numbers, further solar radiation absorption and warming, etc etc. How everything is dead and I how you are feeling happy with yourself.

You could code this modification in – adjusting the (local) value of T that each species of daisy 'sees' (as per the Box and the reference). Or ... we could simply give them different temperature optima, which is what the value of 22.5°C accomplishes in the temperature-dependent growth modifier equation. For now, this is the way-simpler approach and involves only a minimal edit to your existing daisy function. So where in the equation for β_w and β_b you currently have values of 22.5 (°C) in each – try making these different. Reasonable would be to assume that the white daisies are more adapted to hot climates and hence have a higher temperature tolerance, with black daisies being better adapted to colder climates, using their higher albedo and presumably local heating to make up for a colder ambient environment. (You could be able to come up with something not entirely dissimilar to Figure 6.16.)

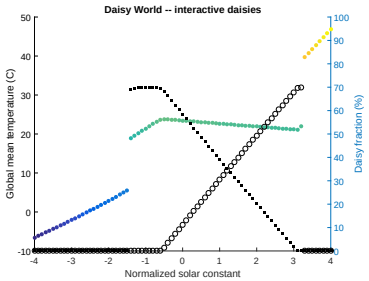

Figure 6.16: Evolution of global surface temperature and the two populations of daisies with time.

7
Dynamic (time-stepping) modelling

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

7.1 *Catch the ball (ballistics and simulating trajectories)*

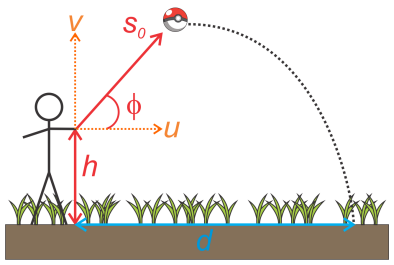

Figure 7.1: Schematic of the thrown-ball system.

In considering dynamic, time-stepping representations of physical (/biogeochemical) systems, we'll start with a simple, ballistics example – that of the trajectory of a thrown ball.

The system we'll consider is shown schematically in Figure 7.1. In essence: we want to determine d – the horizontal distance (m) that the ball travels before it hits the ground. The initial conditions are:

1. The ball is thrown from an initial height h (m).
2. The ball is thrown with an initial speed s_0 (ms^{-1}).
3. The ball is thrown at an initial angle θ with the horizontal.

We'll neglect any air desistence or spin imparter with the ball, and for the purpose of calculating its height, we'll ignore its diameter, i.e. we'll consider that the ball is level with the ground when its centre is at height zero. Over and above this, you'll only need to know the gravitational constant (i.e. gravitational acceleration) – $g = 9.81ms^{-1}$ (i.e. the ball is being thrown on an Earth-like planet near sealevel).

To simply things and the construction of the code and encapsulation of the physics of the model, we'll break it down into 4 steps:

Part I Considering only horizontal travel.

Part II Considering only vertical travel.

Part III Considering both horizonal and vertical travel and testing for when the ball hits the ground.

Part IV Add some graphical output.

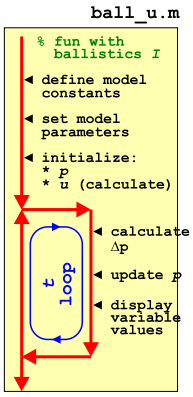

Figure 7.2: Schematic of the code for simulating the horizontal movement of a ball.

Part I Start with a new m-file. Create a structure along the lines of Figure 7.2, i.e. you are going to need to define some constants (g), parameters (the initial height h, initial speed (s_0), initial angle (θ) of the ball).

Because you are going to use a time-stepping approach (rather than solve the system analytically), you are going to need a loop in time, starting at time zero. Can you guess the time-step you need? No? Then we need to make the time-step a parameter that we can change to ensure that the system is solved well (i.e. accurately and without numerical instability). You could call this parameter e.g. dt and set it to an initial (guessed) value[1] such as $0.1s$. How long should you run the simulation for? This is also a sort of unknown at this point, at least until you have run the simulation a couple of times to get a feel for what the longest time the ball stays in the air might

[1] In the parameter section of the code.

be. So why not pick 100s to start with. Again, create a parameter to hold the value of the maximum model simulation time and assign its value in the parameter definition section of the code. Assuming a time-step parameter name of `dt` and a maximum time parameter, `max_t`, if your current time is called `t`, your loop structure will look like:

```
for t = 0:dt:max_t
    %SOME CODE
end
```

with time `t` starting at zero, and progressing to `max_t` in steps of `dt`.

What else do you need? You need a variable to represent the horizontal position of the ball (delineated here in the text as p, with units of m). This will start at zero and be updated within the loop. So also in the parameter section, why not define your horizontal position variable p and assign it a (initial) value of zero.

Lastly, you need to know the horizontal component of the balls' velocity.[2] You can calculate the (initial) horizontal component of velocity from the given initial conditions of initial speed (s_0) and initial angle of trajectory (θ). For now, pick any 'reasonable' values for s_0 [3] and θ [4]. In the figure, the velocity component is designated u.

Along with the schematic of the code structure, this should be all you need to create a basic code (but one at this point that does not actually 'do' anything). You should have a constant defined, and then 5 *parameters* – 3 representing the initial conditions of the model (the parts Figure 7.1 colored in red), plus 2 parameters for the maximum time and time step. You have 3 *variables* in the code so far – time t, which is part of the loop, (horizontal) position p, which you should have initialized to zero, and (horizontal) velocity component u, which you should have initialized calculated from s_0 and θ. There should be nothing in the loop so far.

Check that it runs without error even though it is doing nothing useful! Maybe add some debug (e.g. a line in the loop using `disp`) to check that the loop really does loop from zero to `max_t` in steps of `dt`.[5]

Now to add some code to the loop. In each time-step, i.e. each time around the loop, dt time (s) passes. In time dt, if the horizontal velocity of the ball is u, you should be able to calculate how far it moves, right? You need to add this increment in distance to the current value of the position variable p[6]. Do this.

Re-run the code. Check it works at all (if not: debug). Try adding debug code within the loop that displays the current time (t) plus value of p at each time-step, e.g.

```
for t = 0:dt:max_t
    %CODE TO UPDATE POSITION
```

[2] In the absence of air resistance, horizontal velocity does not actually change throughout the simulation (i.e. in each iteration of the loop, it will have the same value).

[3] On September 24, 2010, against the San Diego Padres, Chapman was clocked at 105.1 mph (169.1 km/h) – the fastest pitch ever recorded in Major League Baseball. If you convert 169.1 km/h into units of ms^{-1}, this will give you some reasonable upper limit for your initial thrown velocity.

[4] Obviously, the angle should lie between zero and 90 °(or else the throw is going backwards and/or into the ground). BE CAREFUL as **MATLAB** assumes that angles are in units of radians, so either work in units of radians throughout, or convert from degrees into radians when you calculate the velocity component based on the angle.

[5] Note that depending on whether or not `max_t` is divisible by `dt` with no remainder, your loop might not exactly finish at a value for `a` of `dt`.

[6] i.e. with code like

```
p = p + delta_p;
```

which you have seen endless times before now and should becoming wearily familiar ...

```
    disp(['current time = ', num2str(t), ', position = ', num2str(p)]);
end
```

so that you can track what is going on. (You can make a fancier output if you wish and add in the relevant units to the output.)

Strictly, when updating the position of the ball in the first iteration of the loop, time is dt at this point, not zero, which is what the loop thinks (you already have a position of zero at time zero – the initial conditions). So rather than starting the loop at zero, make a minor modification and start at a value of dt.

You should have a working model at this point, albeit only for the horizontal position of the ball.

Part II Now for tracking the vertical position (and velocity) of the ball. Copy your previous **m-file** and we can use this as a starting point for the new model.[7]

Think about what is different about the physics of the system (Figure 7.1) from before – this is going to directly inform how you adjust and add to the code. To start with, you should have noticed that the initial position (p) of the ball, does not start at zero, but rather at h. This is one change to make in the code (i.e. having defined h as a parameter, you subsequently use h to set the initial value of p). Also – the initial velocity component, v, is different from before (and in fact is assigned a different letter in Figure 7.1). So change the calculation of the initial velocity component and change the name of whatever variable you used for u to something distinct that you'll remember stands for v in the equation. Overall, the code structure looks like Figure 7.3.

You could, and indeed should, test the code so far. It should in fact do something very similar to before, with position p increasing, linearly, as a function of time (i.e. as the loop progresses in the number of iterations carried out). The only differences you should see are that p starts from value h and the rate at which p changes will be greater or less than before, depending on the value of θ you assumed.[8]

So far so good. Except balls generally do not continue travelling vertically for ever. You are missing gravity in this (vertical-only) model. Your variable for v (vertical velocity) now needs to change as a function of time and you'll need to update its value within the loop[9]. How are you going to update v? Well, the change in velocity with time is called acceleration and in this example the only force exerting any acceleration on the ball is gravity. Mathematically we can approximate the change in velocity, Δv as:

$$\Delta v = -\Delta t \cdot g$$

[7] So for instance we will now interpret p as the vertical, not horizontal position of the ball.

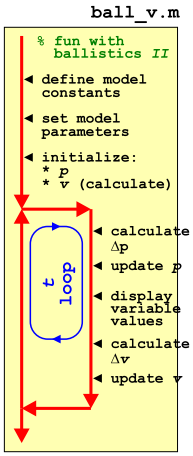

Figure 7.3: Schematic of the code for simulating the horizontal movement of a ball.

[8] What value of θ would result in an identical change in d with time (comparing the previous horizontal-only model with the new vertical (only) one)?

[9] Before or after the updating the position? Actually, a slightly tricky question.

where g is the acceleration due to gravity. Note the appearance of a minus sign in the equation if we are considering a coordinate system with distance upwards.

So in the loop[10] calculate the change in velocity during the time-step, and then update the value of v[11].

Re-run the model ... what happens? Does this seem 'reasonable' ... ? At this point you might consider whether you really do need to run the model for as long as 100s. Play about with the assumed initial angle and also the velocity and get a feel for what is the longest the ball lasts in the air (i.e. until its position becomes negative).

[10] HINT: at the end of the loop.

[11] Hint:

$$v_{(t+1)} = v_{(t)} + \Delta v$$

where $v_{(t+1)}$ is the new (at the next time-step) velocity and $v_{(t)}$ the current velocity

---

Part III You should now have 2 working models (sperate **m-files**) – one for the horizontal position of the ball, and one for the vertical position (and vertical velocity) of the ball. You now want to combine the 2 sperate parts of the model. I suggest basing the combined model on the vertical model (as it is the more complicated of the 2) and hence copying-and-renaming the 2nd script.

How to merge? Mostly, the code content of the 2 individual models was almost identical. What you do need to copy across from the horizontal model is:

- The calculation of the initial value of u.
- The initialization of the horizontal position.
- The calculation of the change in horizontal position each time-step.
- The updating of the new horizontal position.

By now, you should have noted a slight problem – in both previous (sperate) models, the variable h was used to represent both horizontal AND vertical velocity. D'uh!

duh
exclamation informal
used to comment on an action perceived as foolish or stupid, or a statement perceived as obvious. As in:
"I used the same variable name twice – duh!"

My solution would be ... a vector to store the current position – just of one row and two columns, i.e. exactly as you might write a position in (x, y) notation. The horizontal position (x) is hence assigned the first element (`p(1)`) and the vertical position, the 2nd (`p(2)`). If you do this (i.e. resolve the variable clash this way), you'll need to edit how you set the initial conditions in the code, e.g.

```
p(1) = 0;
p(2) = h;
```

as well as how the position is updated in the loop. You can leave the name of the increment in position (Δp) the same if you wish (as this is a temporary variable whose value is replaced each time around the loop in any case).

Hopefully this works and runs ... Maybe add some output within the loop to track its progress, such as:

```
for t = 0:dt:max_t
    %CODE TO UPDATE POSITION
    disp(['(', num2str(p(1)), ',', num2str(p(2)), ') @ t = ', num2str(t)]);
    %CODE TO UPDATE VELOCITY
end
```

You should end up with output, depending on how you constructed the string to be displayed by `disp` (and what initial conditions you chose ...), like:

```
» ball_uv
(0.5,1.866) @ time 0.1
(1,2.634) @ time 0.2
(1.5,3.3038) @ time 0.3
(2,3.8755) @ time 0.4
(2.5,4.3491) @ time 0.5
(3,4.7247) @ time 0.6
(3.5,5.0021) @ time 0.7
(4,5.1814) @ time 0.8
(4.5,5.2626) @ time 0.9
(5,5.2458) @ time 1
(5.5,5.1308) @ time 1.1
(6,4.9177) @ time 1.2
(6.5,4.6065) @ time 1.3
(7,4.1973) @ time 1.4
(7.5,3.6899) @ time 1.5
(8,3.0844) @ time 1.6
(8.5,2.3808) @ time 1.7
(9,1.5792) @ time 1.8
(9.5,0.67938) @ time 1.9
(10,-0.31849) @ time 2
(10.5,-1.4145) @ time 2.1
...
...
```

which is far far far from exciting ... but does at least confirm a constant change in horizontal position with time, and a vertical position that initially increases above the initial condition ($h = 1.0$) but subsequently drops back and eventually falls below zero. And the point at which it reaches zero is the value of d of course.

The very least we could do at this point is to detect when the ball has reached the ground and terminate the loop. I'll leave this code for you to devise, but you'll need:

1. A conditional to test whether the vertical position has dropped below zero. This would go in the loop just after the position of the ball has been updated, And ...
2. The **MATLAB** command to exit a loop, which you have seen before.

Now you might note that when the ball reaches the ground (technically: its height falls below zero) and the loop exists, you may already be way below zero. In fact, if you are even the least little bit observant, you might note that the change in height per time-step at

the end of the simulation is quite large (order meter) and hence it is unlikely you'll ever capture the moment that the ball is very close to the ground. Unless you shorten the time-step, that is. So play about with a shorter time-step (you only need change the value you assigned to the parameter representing Δt in the code). How short does it have to be in order to catch the moment the ball reaches the ground (passes zero) to within e.g. $10cm$?[12] What about $1cm$?

[12] i.e. to have the loop terminate when the height is no more than $-10.0cm$.

Part IV Some graphics fun.

It would be kinda fun (really) to show the ball flying through the air. There are a variety of ways of doing this. We'll start with the simplest first and use `scatter`.

As a departure from previous plotting, we don't want to plot at the very end (after the loop)[13] but rather, plot each position as it is calculated, within the loop.

[13] Although if you stored the position of the ball at each time-step, you could re-play the trajectory afterwards.

First open a new graphics figure window and set `hold` on by adding the lines, before the loop starts:

```
figure;
hold on;
```

Within the loop, you want to plot each (x, y) position as it is calculated (after the position has been updated, that is):

```
scatter(p(1),p(2));
```

(feel free to add additional parameters to `scatter` to make the points smaller or larger, or filled, or whatever). Comment out any debug (`disp`) lines.

Well, not so exciting. The plots sort of appears all at once and there is no sense of animation or of the ball moving. **MATLAB** is just way too fast for its own good[14].

[14] This is a Trump-ism. In truth, **MATLAB** is about the slowest piece of *$&% about.

You can make the loop proceed slower, by adding a time delay – i.e. each time around the loop, **MATLAB** will take whatever time it needs to carry you the calculation and plot the current position PLUS whatever additional time you tell it to chill out for. The command is `pause` and you might initially try e.g.

```
pause(0.05);
```

which should insert a $50ms$ delay into the loop. Run it.

Now it has all got really trippy. If you tell it no different, `MATLAB` insists on auto-scaling the (x and y limits of the) plot. As the position of the ball increases (initially) in y-axis direction, and (constantly) along the *x-axis* direction, **MATLAB** periodically re-scales the axes. Annoying. So before the loop and after you create the figure window, why not prescribe axes limits(?) Having played with the model you

`pause`
MATLAB says: "*`pause(mjs)` pauses the MATLAB job scheduler's queue so that jobs waiting in the queued state will not run.*"
Garbage.
`pause(n)` will pause the execution of the code by `n` seconds.

`axis`
For once, helpfully, MATLAB says: "*`axis([xmin xmax ymin ymax])` sets the limits for the x- and y-axis of the current axes.*"
which is about all you need to know (other than the minimum and maximum limits along the *x-axis* are represented by `xmin`, `xmax`, and the minimum and maximum limits along the *y-axis* are `ymin`, `ymax`).

should have a reasonable idea for what the maximum vertical and horizontal distances are associated with 'reasonable' choices for the initial conditions (s_0 and θ). Don't forget the command for specifying a scale for the axis limits is `axis`. (Figure 7.4-esk maybe?)

Your final task is simply to play about with the pause interval, and the model initial conditions. You can have al the trajectories appearing on the same plot if you comment out the `figure` command in your script, and open a single new figure window at the command line (» `figure`). Then each and every time you run the script, the new trajectory will be added on top. You might also try turning your script into a function so that you do not need to edit the values of s_0 and θ in the code, but pass them into the program as parameters instead (the function needs not return anything however).

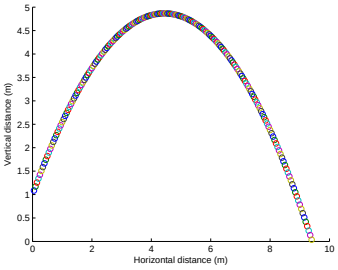

Figure 7.4: Trajectory of a ball!!

7.2 *Dynamics in the zero-D Energy-balance climate model*

We'll now make the zero-D energy-balance climate model (very) slightly more interesting, or at least, (very) slightly more realistic. The time-dependent behavior of the initial version of the energy balance model is trivial. In fact: there isn't any. The system is always in equilibrium as constructed. Why? No thermal inertia – i.e. nothing in the system defined so far has been given any heat capacity and the outgoing (longwave) energy flux is always assumed to be in exact equilibrium with the incoming (shortwave) flux. So we need to add an ocean, or rather: a box (a *variable* in the **MATLAB** code) to store the heat content, or temperature, of the ocean, and update this (temperature) in the event of there being any imbalance between gain and loss of energy at the surface of the Earth.

The science behind the new model is based directly on the basic energy balance equations you had before, except this time you are not going to assume the 2 equations equal (and solve for T) but employ them directly. Instead, you are going to calculate the net energy gain (or loss) over a given interval of time and use the specific heat capacity of a substance (assuming water here)[15] to link the energy change to a temperature change (see Box). This will be the basis of the 'dynamics' of the climate model and will dictate how quickly the mean surface temperature responds to any imbalance in loss vs. gain of energy. You can also assume the following:

- The average mixed layer depth of the ocean is 70 m.
- The average fraction of the Earths surface that is ocean is 0.7.

(both from *Henderson-Sellers* [2014]). You'll also need to know:

- The specific heat capacity of water.

but you can find this out for yourself ...[16] Note that you do not need to know e.g. the radius of the Earth as we are constructing the model on a global average per m^{-2} basis as before.

The form of the program is shown schematically in Figure 7.5 and you'll need to create yourself a new script (`scr_1`) to make this. Much of this and the main sections of code should look familiar. Break the code down into logical sections. Start by defining any constants you need, as well as parameter values. For the time loop, we are going to start off with a fixed total duration and a fixed time step (a little later we'll relax these constraints). And to make things really simple to start – assume a 100 year duration (starting at $T = 1.0$) and a time increment $\Delta T = 1.0$. So you are not even going to need to initialize and update a loop counter in the code! In the loop itself, you firstly need to calculate the energy imbalance (assuming there is

Specific Heat Capacity

According to wikipedia: "*An object's* [or here: ocean] *heat capacity (symbol C) is defined as the ratio of the amount of heat energy transferred to an object and the resulting increase in temperature of the object:*"

$$C = \frac{Q}{\Delta T}$$

where Q is the (change in) energy (so could equally be written ΔQ if you prefer) and ΔT the associated change in temperature. Units are:

- C — JK^{-1}
- ΔT — K
- Q — J

[15] Once again – be very careful with the units. Or all will be lost ...

[16] Be careful to end up with CONSISTENT units!

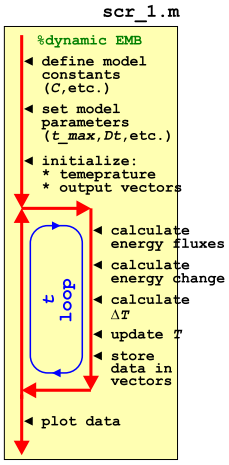

Figure 7.5: Schematic of the script for the basic dynamic EBM

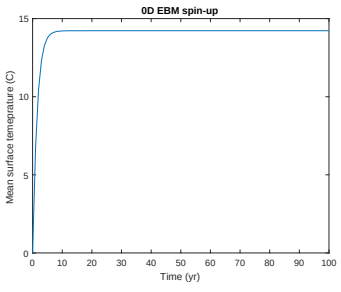

Figure 7.6: 100 yr spin-up of the basic EBM.

one) – remembering that the energy fluxes are in units of Wm^{-2}, i.e. $Js^{-1}m^{-2}$, so you'll need to take the time-step duration into account and find the number of J of heat gain/loss during that time (in s)– then use this to update the temperature of the mixed layer ocean.[17] Then after the loop, plot something helpful at the end.

If successful, you should see something similar to (actually, identical to) Figure 7.6 (assuming a 1 yr time-step).

Next, you are going to play a little with the time-step in the model. So rather than a simple loop from 1 to 100 (years) with an increment of 1, you are going to generalize the increment as Δt. If `dt` is your parameter representing the increment in time (presumably, conveniently defined hear the start of the code)[18], and `max_t` the maximum time (here: 100 years) (also conveniently defined near the start of the code?), then:

```
% start of time-stepping loop
for t = 1:dt:max_t,
    % SOME CODE GOES HERE
end
```

Now however, you will need to crete yourself a loop counter in order to store the results (for subsequent plotting), as because `dt` will not necessarily be an integer, you will not be able to use `a` to index your data storage vector (/array). The modification needed is only minor however – see Figure 7.7. The only slight complication is in knowing the size of the output vectors, assuming that you have created them (using `zeros`) up-front in the code (and as per the Figure 7.5 schematic), rather than growing the vectors as the loop progresses (see earlier). Initially, you would have been able to simply write e.g.

```
data_time  = zeros(100);
data_T = zeros(100);
```

One strategy is simply to pick a number larger than you think the number of times the loop will execute. The downside being that you might create a vast array with only a small portion of it ever being used. Better in this example would be to append to the vectors as the loop progresses and not attempt to define them beforehand (i.e. Figure 7.5 rather than Figure 7.7).

By playing around with different parameter values for Δt, you should discover that some care has to be taken with the choice of time-step duration, e.g. Figure 7.8 has a time-step of 3.5 years, which clearly is on the verge of going doolally. [19]

So far, so far from exciting – you have been simply time-stepping the model to equilibrium, for which there was an analytical solution anyway (with ocean heat capacity irrelevant to this). However, it should be apparent that it takes some years (how many) for the system to reach equilibrium. This would have important implications for

[17] It is much easier and less prone to bug, if you do this in two stages. You could even split things into four:

1. Incoming energy flux.
2. Outgoing energy flux.
3. Net energy change (per m^2) at the Earths surface.
4. Update surface temperature.

[18] Don't forget to convert `dt` into units of s when you use it in the energy calculation.

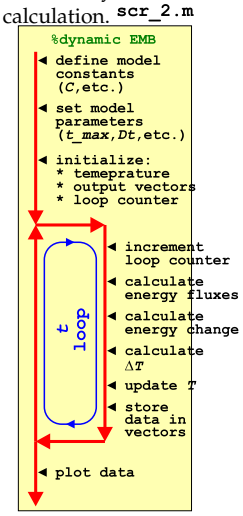

Figure 7.7: Schematic of the script for the basic dynamic EBM – now with added loop count(!)

[19] For practice (fun!?), you could turn the script into a function. Make two parameters as inputs: (1) the total simulation duration, and (2) the time-step, both in units of yr.

Doolally
Mad, insane, eccentric.

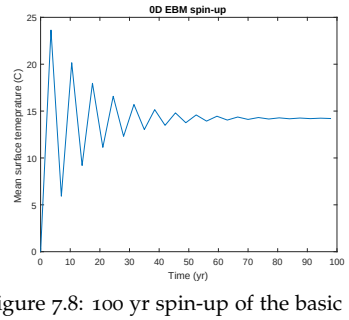

Figure 7.8: 100 yr spin-up of the basic EBM, but with a poor choice of time-step ...

a (real world) system in which the one of the terms in the radiative balance equation changes relatively rapidly (or on a time-scale comparable to the adjustment time of the system). The concentration of CO_2, and radiative forcing due to the 'greenhouse effect', is just such an example.

A follow-on Example to this, takes the time-stepping (dynamic) zero-D EBM (`scr_1`) and drives it with a time history of atmospheric CO_2 concentration (technically: mixing ratio) data.

First off: check out the CO_2 radiative forcing (Greenhouse Effect) Box. This will guide you as to how you are going to modify your energy budget (within the time-stepping loop) – basically, you are simply adding a 3rd term (and a second incoming term) to the heat budget. Test the model first with a fixed, assumed CO_2 concentration and check that the mean surface temperature responds in a reasonable way.[20,21]

The first thing you are going to do, is to take your previous script (`scr_1` or `scr_2`, it does not really matter) and turn it into a function, with a single input (`co2`) and no output. The passed parameter `co2` (or call it something different) will be the concentration of CO_2 in the atmosphere in μatm (equivalent to units of ppm for 1 atmosphere total pressure). You'll then need to edit the calculation of the energy loss/gain by incorporating the greenhouse effect term. The code looks not much different from before – Figure 7.9).

From your previous experiments, you should have determined what value the equilibrium temperature ended up as. You should make this your new initial condition for the planetary temperature and set the appropriate parameter. (If you don't, the results of all your subsequent experiments will be dominated by the climate system adjusting from your initial condition rather than necessarily responding to whatever perturbation you have applied (/experiment carried out).) Having done this, explore the effect of calling your function and passing values for CO_2 different from $278ppm$ ($278\ \mu atm$). For reference:

- Peak of last glacial — $\sim 190ppm$
- Pre-industrial — $278ppm$
- Current — $\sim 400ppm$
- End of century — $\sim 900ppm$
- Cretaceous — $\sim 834 - 1112ppm$(?)

or try other values.

The Greenhouse Effect

The effect of changing CO_2 concentrations on the global energy budget is typically written in terms of a virtual (long-wave) radiation flux applied at the top of the atmosphere. The flux anomaly, ΔF, as a function of CO_2 concentration (technically: mixing ratio) (CO_2) relative to a reference (pre-industrial) concentration (typically: $CO_{2(0)} = 278ppm$) can be approximated:

$$\Delta F = 5.35 \cdot ln(\frac{CO_2}{CO_{2(0)}})$$

The complete basic EBM energy budget now looks like:

$$F_{in} = \frac{\alpha \cdot S_0}{4} + 5.35 \cdot ln(\frac{CO_2}{CO_{2(0)}})$$

$$F_{out} = 0.62 \cdot \sigma \cdot T^4$$

[20] What is 'reasonable'? Well, you could conduct a pair of experiments – one in which you do not modify CO_2, and one in which your double it. The IPCC and there (now) five Assessment reports have much to say about the climate system response to a doubling of CO_2. So you can conduct a reality check on your model based on existing and widely available climate sensitivity information.

[21] By way of reference: assume that the pre-industrial concentration (mixing ratio) of CO_2 in the atmosphere ($CO_{2(0)}$) is 278 ppm.

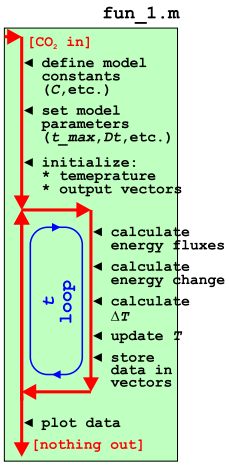

Figure 7.9: Schematic of the dynamic EBM as a function and with the CO2 concentration passed in.

THE FINAL EXAMPLE involves loading in a CO_2 data-set and driving the dynamic zero-D EBM with a changing concentration of CO_2 in the atmosphere.

Go back again to your first dynamic EBM program (`scr_1`). The new version (`scr_3`) will be similar (Figure 7.10). You need to:

1. Add in code to load in the CO_2 dataset. You are going to use the ice-core derived record from week #1 (`etheridge_etal_1996.txt`).
2. From the resulting data array – determine the minimum and maximum years and the total length (number of rows) of the data. All these values might usefully be stored in variables in your code.
3. Create results vectors of the same length. Create one vector for each of: year, CO_2 value, temperature. (Create a single array instead if you prefer.)
4. Edit the time loop such that it runs from the minimum to maximum year (with a time-step of 1 year).
5. In the loop – take the CO_2 value from that year and use it in the calculation of the radiation balance.
6. Also in the loop – save the current year, CO_2 value, and associated calculated temperature. Be careful that indexing of arrays in **MATLAB** always starts at a value of 1. You will either need to derive an index from the current year, or add a loop counter (it is simple to do the former and it takes less lines of code).

When you have this working you should get something like Figure 7.11 (but note that this was done with not quite the same CO_2 dataset ...). If you want to be fancy you can add a horizontal line indicating the pre-industrial equilibrium solution (using `line`).

Finally, the lagged behavior of the climate system (as encapsulated in your EBM) is maybe not obvious as the forcing (CO_2) is varying. Common in model experiments and characterization, is to create artificial and deliberately simplified forcings and perturbations, so as to more readily diagnose the response time and characteristics of a system. Crete an artificial CO_2 data-set, spanning the same time interval as the real data, and at the same frequency, but substitute an idealized CO_2 forcing in which CO_2 stays constant (at 278 ppm) up until year 1999, then at year 2000, increases to 400 ppm, and stays there. The result of such an experiment should look like Figure 7.12.

Other common model scenarios are linear ramps (up, and/or down) and compound increases, such as a 1% per year increase in the concentration of CO_2 (each and every year) starting ca. 1960.

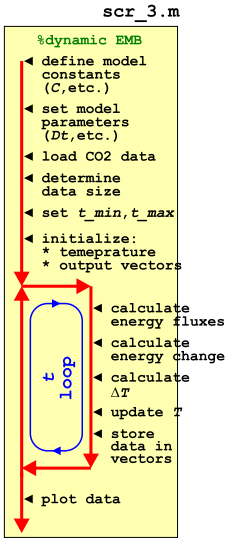

Figure 7.10: Schematic of the dynamic EBM driven by a history of CO2 (read in from a file).

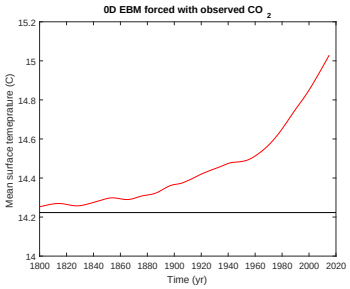

Figure 7.11: Transient EBM response to observed changes in atmospheric CO2. For reference, the pre-industrial equilibrium global temperature is shown as a horizontal black line.

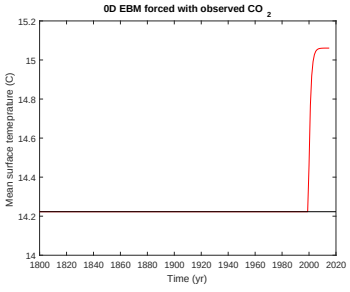

Figure 7.12: Transient EBM response to (fake) changes in atmospheric CO2.

Bibliography

Index

.mat environment, 33
; environment, 18
= environment, 17, 18

addition environment, 18
addpath environment, 32
and environment, 19
assignment operator environment, 19
axis environment, 26, 147

break environment, 66

cell array environment, 78
cell2mat environment, 77, 78
clabel environment, 86, 87
clear all environment, 20
clear environment, 20
close environment, 20
colon operator environment, 22–24, 29
colorbar environment, 88, 99
colormap environment, 85
Command Window, 14
comments environment, 78
contour environment, 84, 86
contourf environment, 84

disp environment, 44, 45, 63
division environment, 18
duh environment, 145

else environment, 52
elseif environment, 52
end environment, 23, 24
environments
  .mat, 33
  ;, 18
  =, 17, 18

addition, 18
addpath, 32
and, 19
assignment operator, 19
axis, 26, 147
break, 66
cell array, 78
cell2mat, 77, 78
clabel, 86, 87
clear, 20
clear all, 20
close, 20
colon operator, 22–24, 29
colorbar, 88, 99
colormap, 85
comments, 78
contour, 84, 86
contourf, 84
disp, 44, 45, 63
division, 18
duh, 145
else, 52
elseif, 52
end, 23, 24
equality, 19
exist, 67, 70
exit, 20, 134
exponentiation, 18
fclose, 76
figure, 25
find, 94, 95, 97
fliplr, 23, 30
flipud, 23
flipup, 30
fopen, 76–78
for, 58
fprintf, 33
functions, 20
geoshow, 91

getframe, 65
greater than, 19
greater than or equal to, 19
hist, 41
hold, 38
if ... end, 52
image, 42, 84
imagesc, 84
imread, 42
inequality, 19
input, 52, 53, 70
legend, 38
length, 23, 72
less than, 19
less than or equal to, 19
line, 132, 152
load, 32–34
ls, 32
m-file, 44
m-files, 26
meshgrid, 88
mod, 129
movie2avi, 65
multiplication, 18
NaN, 97
not, 19
num2str, 64
or, 19
pause, 147
pcolor, 42, 71
pi, 20
plot, 25
print, 27
rocker, 129
rotate, 30
save, 33
scatter, 25, 39
set, 107
sin, 26

size, 23, 29
sort, 34
sortrows, 34
strcmp, 53
subplot, 27
subtraction, 18
sum, 30
switch ... case ... end, 57
textscan, 76–78
title, 26
transpose, 30
transpose operator, 23
while, 58
xlabel, 26
xlsread, 80
ylabel, 26
zeros, 128
equality environment, 19
exist environment, 67, 70
exit environment, 20, 134
exponentiation environment, 18

fclose environment, 76
figure environment, 25
find environment, 94, 95, 97
fliplr environment, 23, 30
flipud environment, 23
flipup environment, 30
fopen environment, 76–78
for environment, 58
fprintf environment, 33
functions environment, 20

geoshow environment, 91
getframe environment, 65
greater than environment, 19
greater than or equal to environment, 19

hist environment, 41
hold environment, 38

if ... end environment, 52
image environment, 42, 84
imagesc environment, 84
imread environment, 42
inequality environment, 19
input environment, 52, 53, 70

legend environment, 38
length environment, 23, 72
less than environment, 19
less than or equal to environment, 19
license, 2
line environment, 132, 152
load environment, 32–34
ls environment, 32

m-file environment, 44
m-files environment, 26
meshgrid environment, 88
mod environment, 129
movie2avi environment, 65
multiplication environment, 18

NaN environment, 97
not environment, 19
num2str environment, 64

or environment, 19

pause environment, 147
pcolor environment, 42, 71
pi environment, 20
plot environment, 25
print environment, 27

rocker environment, 129
rotate environment, 30

save environment, 33
scatter environment, 25, 39
set environment, 107
sin environment, 26
size environment, 23, 29
sort environment, 34
sortrows environment, 34
strcmp environment, 53
subplot environment, 27
subtraction environment, 18
sum environment, 30
switch ... case ... end environment, 57

textscan environment, 76–78
The command line, 14
title environment, 26
transpose environment, 30
transpose operator environment, 23

variable, 16

while environment, 58

xlabel environment, 26
xlsread environment, 80

ylabel environment, 26

zeros environment, 128