

ANDY RIDGWELL

STR = 'DO YOU LIKE BANANAS?'

Copyright © 2019 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/>)

*Current printing, December 2019*

# Contents

	<i>How to use this Textbook</i>	17
	0.1 <i>Fonts and highlighting</i>	17
	0.2 <i>Help(!) and keyword definitions</i>	17
	0.3 <i>Side notes and other distractions from the main text</i>	18
	0.4 <i>What and when to type</i>	18
	0.5 <i>Code structure</i>	19
	0.6 <i>'Answer' codes</i>	20
1	<i>Elements of ... MATLAB and data visualization</i>	21
	1.1 <i>Using the MATLAB software</i>	22
	1.1.1 <i>Starting MATLAB</i>	22
	1.1.2 <i>The command line</i>	22
	1.1.3 <i>MATLAB GUI</i>	23
	1.1.4 <i>Help(!)</i>	23
	1.2 <i>Basic concepts</i>	24
	1.2.1 <i>Variables</i>	24
	1.2.2 <i>Numerical expressions and Arithmetic operators</i>	27
	1.2.3 <i>Relational and logical operators</i>	28
	1.2.4 <i>Functions (built-in)</i>	29
	1.2.5 <i>Miscellaneous commands</i>	29
	1.3 <i>Vectors and arrays #1</i>	31
	1.3.1 <i>Creating vectors</i>	31
	1.3.2 <i>Basic vector manipulation</i>	32
	1.3.3 <i>Addressing elements in vectors</i>	32

1.4	<i>Basic graphing (aka. 'data visualization')</i>	34
1.4.1	<i>Plotting</i>	34
1.4.2	<i>Graph labelling</i>	35
1.4.3	<i>Sub-plots</i>	36
1.4.4	<i>Saving graphics and figures</i>	36
1.5	<i>Vectors and arrays #2</i>	38
1.5.1	<i>Creating matrices and arrays</i>	38
1.5.2	<i>Basic matrix manipulation</i>	39
1.5.3	<i>Some matrix math :(</i>	41
1.6	<i>Loading and saving data</i>	43
1.6.1	<i>Where am I?</i>	43
1.6.2	<i>Loading and importing data</i>	44
1.6.3	<i>Saving and exporting data</i>	45
1.6.4	<i>Loading and saving the workspace</i>	45
1.7	<i>Basic data processing (and yet more plotting)</i>	47
1.7.1	<i>Sorting data (in arrays)</i>	47
1.7.2	<i>Data scaling</i>	49
1.7.3	<i>Data (row) deletion</i>	50
1.8	<i>Nicer graphing</i>	53
1.8.1	<i>Modifying lines/symbols in plot</i>	53
1.8.2	<i>Plotting multiple data-sets</i>	53
1.8.3	<i>Changing label font size (and type)</i>	54
1.8.4	<i>Scatter plots</i>	55
1.8.5	<i>Simple 2D data and bitmap visualization</i>	55
1.9	<i>Further matrix math (systems of equations)</i>	57
2	<i>Elements of ... programming</i>	61
2.1	<i>Introduction to scripting (programming!) in MATLAB</i>	62
2.1.1	<i>Programming good practice</i>	63
2.1.2	<i>Debugging the bugs in buggy code</i>	65



2.2	<i>Functions</i>	68	
2.3	<i>Conditionals '101'</i>	71	
2.3.1	<i>if ...</i>	71	
2.3.2	<i>switch ...</i>	76	
2.4	<i>Loops '101'</i>	78	
2.4.1	<i>for ...</i>	78	
2.4.2	<i>Other loop configurations and usages</i>	82	
2.4.3	<i>Fun(!) worked examples</i>	83	
2.5	<i>Loops and conditionals ... together(!)</i>	89	
2.5.1	<i>for ... and conditionals</i>	89	
2.5.2	<i>while ...</i>	92	
2.6	<i>Even more (and loopier) loops</i>	95	
3	<i>Further ... MATLAB and data visualization</i>	99	
3.1	<i>Further data input</i>	100	
3.1.1	<i>Formatted text (ASCII) input</i>	100	
3.1.2	<i>Importing ... Excel spreadsheets</i>	104	
3.1.3	<i>Importing ... netCDF format data</i>	105	
3.2	<i>Further (spatial / (x,y,z)) plotting</i>	108	
3.2.1	<i>Contour plotting</i>	108	
3.3	<i>Further data processing</i>	117	
3.3.1	<i>find!</i>	118	
3.3.2	<i>Other data filtering</i>	123	
3.3.3	<i>Basic (pretend) 'stats'</i>	124	
3.3.4	<i>Some useful data manipulations techniques</i>	126	
3.3.5	<i>Data interpolation</i>	127	
3.4	<i>Even nicer graphing and graphics</i>	131	
3.4.1	<i>Drawing lines and shapes</i>	132	
3.4.2	<i>Colors</i>	137	
3.4.3	<i>Placing and making text 'nice'</i>	137	
3.4.4	<i>Creating color maps</i>	138	

4	<i>Further ... Programming</i>	141
4.1	<i>Nested loops</i>	142
4.2	<i>Algorithms and problem-solving</i>	150
4.2.1	<i>Example #1: max(!)</i>	150
4.2.2	<i>Example #2: sort(!)</i>	155
4.2.3	<i>Example #3: a gridded problem</i>	158
4.3	<i>Interpreting equations (0) – Basics</i>	170
4.4	<i>Interpreting equations (1) – Population models</i>	171
4.4.1	<i>Exponential (and unrestricted) growth</i>	171
4.4.2	<i>Restricted growth (and an equilibrium state)</i>	172
4.5	<i>Interpreting equations (2) – Pure lovely maths</i>	175
4.5.1	<i>Sequence convergence (in 1D)</i>	175
4.5.2	<i>Sequence convergence (in 2D)</i>	178
5	<i>Programming applications – games!</i>	185
5.1	<i>Tic-tac-toe</i>	186
5.1.1	<i>Mouse behavior</i>	189
5.1.2	<i>Drawing the 'objects'</i>	189
5.1.3	<i>Identifying specific boxes</i>	191
5.1.4	<i>Remembering turns (and arrays!)</i>	193
5.1.5	<i>Putting it all together</i>	194
6	<i>Numerical modelling – zero-D / equilibrium</i>	199
6.1	<i>Zero-D Energy-balance model of the climate system</i>	200
6.1.1	<i>The basic EBM</i>	201
6.1.2	<i>The EBM as a function</i>	203
6.1.3	<i>Creating a function for the evolution of solar constant through geological time</i>	204
6.1.4	<i>Using multiple functions and calculating global surface temperature as a function of geological time</i>	205
6.1.5	<i>Parameter sensitivity experiments using the EBM – #1</i>	208
6.1.6	<i>Parameter sensitivity experiments using the EBM – #2</i>	211

6.2	<i>'Daisy World'</i>	214
6.2.1	<i>'fixed daisy' daisy-world</i>	215
6.2.2	<i>'dumb daisy' daisy-world</i>	217
6.2.3	<i>'clever daisy' daisy-world</i>	221
6.2.4	<i>Efficient and 'clever daisy' daisy-world</i>	222
7	<i>Numerical modelling – Dynamic (time-stepping)</i>	225
7.1	<i>Catch the ball (ballistics and simulating trajectories)</i>	230
7.2	<i>Dynamics in the zero-D Energy-balance climate model</i>	240
8	<i>Numerical modelling – To infinity (1D) and beyond(!)</i>	245
8.1	<i>1-D energy-balance climate model</i>	246
8.2	<i>1-D reaction-transport model</i>	252
9	<i>Graphical User Interfaces (GUI)</i>	263
9.1	<i>MATLAB GUI basics</i>	264
9.1.1	<i>Hello, World [Static Text (box)]</i>	266
9.1.2	<i>Simple GUI responses [Push Button]</i>	268
9.1.3	<i>Updating object properties (do you like bananas?)</i>	271
9.1.4	<i>Simple GUI responses [Sliders]</i>	275
9.2	<i>MATLAB apps</i>	277
10	<i>Numerical modelling meets GUI (prettier games!)</i>	279
10.1	<i>GUI Pokémon game</i>	280
11	<i>Example codes</i>	297
11.1	<i>Chapter 1 codes</i>	298
11.2	<i>Chapter 2 codes</i>	299
11.3	<i>Chapter 4 codes</i>	301
11.4	<i>Chapter 10 codes</i>	302

*Bibliography* 303

*Index* 305

# List of Figures

1	Schematic for a generic <i>script</i> .	19
2	Schematic for a generic <i>function</i> .	19
1.1	Example of the default output of the <code>plot</code> function.	34
1.2	A plot illustrating axis auto-scaling (maximum $x$ and $y$ values now slightly larger than 10 and 100, respectively).	35
1.3	A (only very slightly) improved plot.	35
1.4	Arrangement of subplots.	36
1.5	Result of simply throwing the entire data matrix at <code>plot</code> ....	45
1.6	Spline fit to measured changes in CO <sub>2</sub> concentration in Law Dome ice core, following <i>Etheridge et al.</i> [1996].	45
1.7	proxy reconstructed past variability in atmospheric CO <sub>2</sub> .	47
1.8	Proxy reconstructed past variability in atmospheric CO <sub>2</sub> (sorted data).	48
1.9	Observed annual global mean surface temperature anomaly (compared to year 1910 to 2000 average).	49
1.10	Observed annual global mean surface temperature.	50
1.11	Observed annual mean surface temperature in Riverside.	51
1.12	Observed global annual mean surface temperature anomaly, relative to the mean of 1910 through 2000.	51
1.13	Observed annual mean surface temperature anomaly, relative to the mean of 1910 through 2000, at Riverside.	52
1.14	Observed annual mean surface temperature anomaly, relative to the mean of 1910 through 2000, at Riverside, filtered to remove years with missing monthly data.	52
1.15	Proxy reconstructed past variability in atmospheric CO <sub>2</sub> (sorted data).	53
1.16	Proxy reconstructed past variability in atmospheric CO <sub>2</sub> (sorted data).	54
1.17	Proxy reconstructed past variability in atmospheric CO <sub>2</sub> (scatter plot).	55
1.18	Proxy reconstructed past variability in atmospheric CO <sub>2</sub> (scatter plot).	55
1.19	A 2D plot of some random gridded model data.	55
1.20	A 2D plot of some random gridded model data ... but with the underlying data matrix re-orientated before plotting.	56
1.21	Lake volumes and river flow rates in the Great Lakes system.	57
2.1	Schematic of the example program.	64

- 2.2 Schematic of the Hello World program. 65
- 2.3 Output from the (bug-fixed version of) `plot_some_dull_stuff m-file`. 67
- 2.4 Schematic structure of the simple bananas question program. 72
- 2.5 Schematic structure of the extended bananas question program. 73
- 2.6 A slight variant on the schematic structure of the extended bananas question program. 74
- 2.7 Schematic of the bananas program using the `if ... else ...` construct (and displaying alternative messages). 74
- 2.8 Extremely unappealing blocky plot of Earth surface temperature (who cares with month? – the graphics are too poor to matter ...). 86
- 2.9 Continental outline (of sorts). 96
- 2.10 Another continental outline (of sorts). 96
- 2.11 Another go at the continental outline! 98
  
- 3.1 Very basic imaging (`image`) of an array (2D) of data – here, global bathymetry. 109
- 3.2 Slightly improved very basic imaging (`imagesc`) of bathymetry data. 109
- 3.3 Example result of basic usage of the `contour` function. 110
- 3.4 Example usage of `contourf`, with the hot `colormap` (giving dark-/brown colors as deep ocean, and light/white as high altitude). 110
- 3.5 Example usage of `contour`, contouring only the zero height isoline, and providing a label. 111
- 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). 114
- 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. 116
- 3.8 Proxy reconstructed past variability in atmospheric CO<sub>2</sub> (scatter plot). 121
- 3.9 Figure window with axes. 132
- 3.10 Figure window with single line segment (via `plot`). 133
- 3.11 Figure window with a second line segment (via `line`). 133
- 3.12 (no comment). 133
- 3.13 Proxy reconstructed past variability in atmospheric CO<sub>2</sub> (scatter plot). 135
- 3.14 Square. 136
- 3.15 Alt square. 136
- 3.16 Random polygon. 136
- 3.17 RGB scale. By SharkD - Own work, GFDL, <https://commons.wikimedia.org/w/index.php?curid=3375025> 137
- 3.18 Global topography plotted with the default **MALTB** color scheme. 138
- 3.19 Global topography plotted with `hot`. 139
- 3.20 Global topography plotted with a basic black+white dual color scheme. 139

- 3.21 Comparison of sparsely sampled data (points) compared with a more finely spaced spline interpolation (solid line). (x-axis and y-axis are both unit-less.) 139
- 3.22 Global topography plotted with a user-defined grey-scale. 140
  
- 4.1 Tic-tac-toe game grid. 142
- 4.2 Tic-tac-toe game grid with numerical codes overlain. 142
- 4.3 Tic-tac-toe game grid – numerical representation. 142
- 4.4 Tic-tac-toe game grid – search order: columns then rows. 143
- 4.5 Tic-tac-toe game grid – search order: rows then columns. 143
- 4.6 3x3 grid of black squares ... 147
- 4.7 3x3 grid of colored squares. 147
- 4.8 (yawn) 147
- 4.9 Chess board grid pattern. 149
- 4.10 Ocean topography (blues through red) in the 'GENIE' Earth system model. Land is shown marked in brown. 158
- 4.11 The 'GENIE' mode land grid, with land points assigned a sequential integer (working across and down the grid – from West to East, and then North to South). 164
- 4.12 The 'GENIE' mode land grid, with land points assigned a unique identifier ... almost ... (!) 167
- 4.13 The 'GENIE' mode land grid, with land points assigned a unique identifier (color). 168
- 4.14 The 'GENIE' mode land grid, with land points (almost) assigned a unique identifier (color). 169
- 4.15 The 'GENIE' mode land grid, with land points assigned a unique identifier (color). 169
- 4.16 The Mandelbrot Set – points representing complex numbers that are members of the set, are shown in black. Complex numbers for which the sequence does not converge, are graphically represented by the white locations in the plotted domain. 175
- 4.17  $\times 50$  (-ish) zoom in on the Mandelbrot Set illustrating self-similarity and the fractal nature of the set boundary. 175
- 4.18 Solution space (blue points) for the simple sequence. 176
- 4.19 Solution space (blue points) for the simple sequence, with the rate of divergence forming the color scale of light blue (slowest) through yellow (fastest divergence). 178
- 4.20 Simple, low resolution Mandelbrot set rendition. 182
- 4.21 Simple, low resolution Mandelbrot set rendition (now highlighting points that are members of the solution set (black) vs. not (white). 182
- 4.22 Initial Mandelbrot Set magnification. 183
- 4.23 Example Mandelbrot Set zoom. 183
- 4.24 Example Mandelbrot Set zoom. 183

- 5.1 Tic-tac-toe. By Symodeo9 - Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=2064271>. 186
- 5.2 Schematic structure of the complete code. 187
- 5.3 Tic-tac-toe game grid drawn. 188
- 5.4 Tic-tac-toe game – object drawing test. 191
- 5.5 Tic-tac-toe game – object drawing + mouse button test. 191
- 5.6 Tic-tac-toe game – object drawing now arranged in a grid. 192
- 5.7 Tic-tac-toe game grid with numerical codes overlain. 193
- 5.8 Tic-tac-toe game – object drawing now arranged in a grid and with forced alternation in player turn. 195
- 5.9 Linear indices of a  $3 \times 3$  matrix. 197
- 6.1 The pattern of absorption bands generated by various greenhouse gases and aerosols (lower panel) and how they impact both incoming solar radiation (upper left) and outgoing thermal radiation from the Earth's surface (upper right). (Figure prepared by Robert A. Rohde for the Global Warming Art project.). 201
- 6.2 Form of the basic EBM model. 202
- 6.3 Form of the basic EBM model as a *function*. 203
- 6.4 Schematic structure of code for calculating the solar constant (output) as a function of time (input). 204
- 6.5 Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, and solar constant and EBM functions. Note – in this schematic, the code contents of the two *functions* remain in their respective **m-files**. The *function* code does not get copy-pasted into the `scr_4.m` script file. The red arrows indicate the passing of variable values ... from `scr_4.m` into each *function*, with the *functions* returning variable values back to `scr_4.m`. 206
- 6.6 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). 206
- 6.7 Schematic structure of the model configured to carry out a single parameter sensitivity study. 211
- 6.8 Sensitivity of global mean surface temperature vs. solar constant (mean surface albedo held constant at an albedo value of 0.3). 211
- 6.9 Schematic structure of the model configured to carry out a double (in terms of solar constant AND now albedo) parameter sensitivity study. 212
- 6.10 Global mean surface temperature (°C) as a function of solar constant and surface albedo grid point number. 213
- 6.11 Global mean surface temperature (°C) as a function of the value of solar constant and surface albedo. 213
- 6.12 Daisy World 214



- 6.13 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. 215
- 6.14 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). 217
- 6.15 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 from the start of the outer loop as before. 219
- 6.16 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. 221
- 6.17 Evolution of global surface temperature and the two populations of daisies with time. 222
  
- 7.1 Schematic of the thrown-ball system. 230
- 7.2 Schematic of the code for simulating the horizontal movement of a ball. 230
- 7.3 Schematic of the code for simulating the vertical movement of a ball. 232
- 7.4 Trajectory of a ball!! 237
- 7.5 Trajectory of a ball (with a poor time-step choice). 238
- 7.6 Trajectory of a ball (even poorer time-step choice). 238
- 7.7 Schematic of the dynamic EBM. 240
- 7.8 Schematic of the script for the basic dynamic EBM 240
- 7.9 100 yr spin-up of the basic EBM. 241
- 7.10 Schematic of the script for the basic dynamic EBM – now with added loop count(!) 241
- 7.11 100 yr spin-up of the basic EBM, but with a poor choice of time-step ... 242
- 7.12 Schematic of the dynamic EBM driven by a history of CO<sub>2</sub> (read in from a file). 243
- 7.13 Transient EBM response to observed changes in atmospheric CO<sub>2</sub>. For reference, the pre-industrial equilibrium global temperature is shown as a horizontal black line. 244
- 7.14 Transient EBM response to (fake) changes in atmospheric CO<sub>2</sub>. 244

- 8.1 Basic 1-D EBM with no latitudinal heat transport and for a single hemisphere only. 248
- 8.2 Basic 1-D EBM with no latitudinal heat transport (red filled circles). Overlain is the zonal mean observational data for January (blue circles). 249
- 8.3 As per Figure 8.2 but for July. 249
- 8.4 1D EBM with an initial guess as to the value of  $k$ . 251
- 8.5 1D EBM with a x10 larger value of  $k$ . 251
- 8.6 Idealized schematic of the soil-CH<sub>4</sub> system. 252
- 8.7 Slightly less idealized schematic of the soil-CH<sub>4</sub> system. 252
- 8.8 Even less idealized and almost realistic, schematic of the soil-CH<sub>4</sub> system. 252
- 8.9 Soil profile of CH<sub>4</sub> after 10.0s of simulation. 259
- 8.10 Soil profile of CH<sub>4</sub> after 100.0s of simulation. 259
- 8.11 Soil profile of CH<sub>4</sub> after 100.0s of simulation with an extremely marginal choice of time-step length. 260
- 8.12 Soil profile of CH<sub>4</sub> after 100.0s of simulation, with CH<sub>4</sub> uptake at the base of the profile with a rate constant of 1.0 per s. 261
- 8.13 Equilibrium soil profile of CH<sub>4</sub>, with CH<sub>4</sub> uptake throughout the soil column with a rate constant of 0.1 per s. 261
- 8.14 Example equilibrium soil profile of CH<sub>4</sub> with production at depth. 262
  
- 9.1 Starting GUI window of the **MATLAB GUIDE**, GUI design tool. 264
- 9.2 (Blank) GUI window editor GUI window. 265
- 9.3 Design of the Hello, World window! 266
- 9.4 Design window with a default push button object. 268
- 9.5 (completely) Bananas design window. 271
- 9.6 (completely) Bananas GUI in action. 273
  
- 10.1 Screen-shot of he Pokémon game App. 280
- 10.2 Trajectory model, with a Pokéball image replacing the scatter point. Here show without deleting the image once displayed. 282
- 10.3 Trajectory model (exactly the same trajectory as per the Figure 10.2), frozen mid-flight at  $t = 1$ s with the Pokéball passing over UC-Riverside. 283
- 10.4 Template App with background image. 288
- 10.5 Template App with background image plus Pokémon. 289
- 10.6 Template App with background image plus small Pokémon at bottom right. 289
- 10.7 Template App with background image plus small Pokémon at bottom right, now with its transparency applied. 290
- 10.8 App with ball trajectory trail. 292

## *List of Tables*

- 1.1 Pollution input rates to each of the 5 lakes. 57
- 4.1 Examples of applying the equation iteratively (different starting values). 176



# How to use this Textbook

A brief guide as to how to interpret and make best use of this book, follows.

## 0.1 Fonts and highlighting

Throughout ... but also be aware (because it is probably not implemented particularly consistently ...): the following formatting is used in the text to distinguish the specific context of the word:

- **Bold** – indicates program/software names (e.g. **MATLAB**).
- *Italics* – indicates technical/jargon words, particularly specific to **MATLAB** (but not command words or functions themselves) or programming concepts, e.g. *loop*.
- Sans-serif font family typeface – indicates keyboard keys (e.g. F5), program menu items (e.g. Save as ...), program window names, and filenames (except where they appear in **MATLAB** code).
- Typewriter font family typeface – indicates **MATLAB** commands and *functions*, and lines of code (see examples below).
- Color highlights in the text are used to reflect the colors employed by **MATLAB** at the command line, or in the code editor.
- Math is hi-lighted in a different font, e.g:

$$a = 10 \times b + c^2$$

and hence differs from the **MATLAB** code version:

$$a = 10*b + c^2$$

or writing it out 'normally':

$$a = 10 \times b + c^2$$

## 0.2 Help(!) and keyword definitions

**MATLAB** help is not always especially helpful! In the course text, for each *function* that **MATLAB** provides a comprehensive help text on, such as `help`, a simple summary version will be displayed in the right hand margin in a grey box. For example – the box headed

**FUNCTION.**

### FUNCTION

A simple and/or summary usage of particular **MATLAB** commands and *functions* is provided in a grey-background box in the margin.

...  
...

Also appearing in grey boxes in the margin are overviews and summaries of **MATLAB** commands or functions as well as ways to do things in **MATLAB**. For example – the box headed *loops*.

*loops*

There are a number of different ways of constructing *loops* in **MATLAB** ...

...  
...

0.3 *Side notes and other distractions from the main text*

<sup>1</sup> sort of things will appear in the text – side notes<sup>2</sup> and there will be some corresponding text or comment in the margin (as closely aligned vertically as possible). Most side notes are helpful and offer additional guidance or suggestions, and on balance, you should read them.<sup>3</sup> In fact, the format of the book gives over substantial space to Side notes, explanation boxes, and figures, so be prepared that important information may frequently appear in the margins.

<sup>1</sup> I am a Side note!

<sup>2</sup> I am also a Side note!

<sup>3</sup> Some are trivial and a little worthless educationally, but you won't know which is which until you have read them ... They might also just brighten up your day a little.

0.4 *What and when to type*

Examples of **MATLAB** code/commands are indicated by text in a 'Typewriter' font, e.g.

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

When the given examples additionally illustrate how they are typed in, and/or, requires you to actually type in the lines at the command line, the text again appears in the 'Typewriter' font, but in addition, the command line prompt (») is shown at the start of a line (you do not actually type in the prompt itself ...), e.g.

```
» hello
```

is asking you to type in hello at the command line, and

```
» hello
Undefined function or variable 'hello'.
```

is then showing you what happens (you to type in hello at the command line)!

---

Lines of code that goes with the discussion in the text and which is not necessarily intended for you to type in (although you may still want to, simply to try it out), is given in a light Courier font:

```
% light font lines of code
```

Lines of code that are intended for you to type in – either at the command line ...

```
» disp('hello')
```

or place in an **m-file** ...

```
% place in a file
```

are given in a **bold Courier font**. Additionally, code to type in, where possible/appropriate, will include the same context-colors as **MATLAB**.

Instructions where you should do or try something out, rather than read and digest, where possible are given in **bold**. (Note that you might want to try out other (light font) code to get a complete picture of the art of programming.)

When you see a string or variable name in all CAPITAL LETTERS – this is a ‘placeholder’ and is indicating that you should substitute in an appropriate string or variable name in its place, e.g.

```
load('FILENAME', '-ascii');
```

is in fact indicating that you substitute the name of your actual file in place of `FILENAME`. i.e., if your actual filename was `exciting_data.txt`, then your code would read:

```
load('exciting_data.txt', '-ascii');
```

Alternatively:

```
plot(MYARRAY(:,1), MYARRAY(:,2));
```

would indicate that you should substitute your actual variable name (holding the data to plot in this example) in place of `MYARRAY`.

In general, you should use all lower-case characters for names of *variables*, *functions* and *scripts*, or files.

## 0.5 Code structure

A visual guide to the structure of your programs is given by schematic figures in the page margin<sup>4</sup>. For example, a generic *script* (yellow box) is shown by **Figure 1**, and a generic *function* (green box) by **Figure 2**.<sup>5</sup>

In these schematics, the flow (sequence) of the code is indicated by the red arrow.

For the *function*, that information is passed into the *function*, and then returned back to where the function was called from, is indicated by the red arrows entering the top of the box and leaving the bottom of the box, respectively. (But note that there is no line of code at the end that tells the model to return values ... this is simply to illustrate the flow of the program, particularly when things get more complicated and there are multiple *scripts* and *functions* involved.)<sup>6</sup>

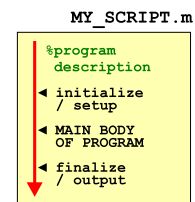


Figure 1: Schematic for a generic *script*.

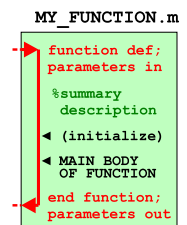


Figure 2: Schematic for a generic *function*.

<sup>4</sup> Not all code fragments and programs are given a schematic.

<sup>5</sup> Don't worry about the terms *function* and *script* for now ...

<sup>6</sup> All this should hopefully all become apparent later.

For the *script*, the code file starts with a comment (`%program description`) summarizing what the *script* does, although after the *function* definition header line, so to should the *function* (somewhere have comment lines describing what it does).

The black left-pointing filled triangles and associated text to the right, indicate categories of code content, and occurring in what order, that the programs might contain.

The purpose of these cartoons is to help you when faced with a blank page and the question: 'Where do I start' or 'What do I write' appears prominently in your mind<sup>7</sup>. It is to give you some sort of idea what bits might go where, and what general content is required in the file. The cartoons do not (and are not intended) to show the exact details of the code content. Nor do they necessarily indicate all the different sections needed. Conversely, not all the sections illustrated may be strictly necessary and in some examples there may be nothing to 'initialize' and there may be no constants or local parameters to define the values of at the program start.

So please – use the cartoons as a simple visual guide to the approximate structure of your program, but do not over-interpret them.

<sup>7</sup> Also surrounded by flashing neon lights.

## 0.6 'Answer' codes

For some of the more complex codes you will be expected to write, in addition to step-by-step instructions in the text, complete 'answer' codes will be provided at the back of the text. These are provided as guides to help you structure the code and see the 'bigger picture' of where all the parts fit together. The complete codes are obviously NOT provided for you simply to copy ... else you'll learn nothing. Except how to use the CTRL-C and CTRL-V key combinations.

Please use this provision as intended and for guidance only should you find yourself completely stuck.



# 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' and what the heck you'd actually do with it. Specifically, you are going to learn about variables and arrays and doing some very basic/simple math in **MATLAB**, and learn how to import and manipulate (array) data in this software environment and then do some basic plotting (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**. And way way better than potato printing.

```
22 str = 'do you like bananas?'
```

## 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*<sup>1</sup>. 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 'Fortnite Battle Royale', 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 graduate and spend the rest of your days doing a proper job? If so, read on ...

<sup>1</sup> 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 nothing (the page is blank) ... or in older versions of the software:

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

Regardless, there should be a vertical blinking line (cursor) following the double 'greater than' symbols with an ocean of blank lines/space below<sup>2</sup>.

If you are unfamiliar with using command-line driven software ... Don't Panic!<sup>3</sup> Nothing bad can happen, regardless of what you do. Well, almost. It is possible to accidentally 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.)

To convince yourself that nothing dreadful will happen ... type ... anything. Actually 'anything' will do.

```
» anything
Undefined function or variable 'anything'.
```

<sup>2</sup> 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* it 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.

<sup>3</sup> Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Pocket Books, 1979. ISBN 0-671-46149-4

Well ... not so exciting. But not so disastrous! **MATLAB** simply has no clue what you are talking about, or rather, anything is not a 'key word'<sup>4</sup> that **MATLAB** recognises. In the specific error message, **MATLAB** could not find that anything was a built-in (or user-defined) *function*, nor a listed *variable*, both of which you'll learn about in due course.

### 1.1.3 MATLAB GUI

There are lots of fancy looking icons and pretty colors and you could spend all day staring at them and not getting any work done. Or you could learn some 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 Graphical User Interface (*GUI*). Look at this as a lesson for the user (to read the *Help*, textbook, on-line documentation, or simply go Google for an answer<sup>5</sup>).

### 1.1.4 Help(!)

If stuck at any point – you can press the F1 key or click on the question mark icon on the tool-bar, to bring up the indexed and searchable **MATLAB** documentation.<sup>6</sup>

You can also type `help` at the command line (and press the Return key).

```
» help
```

The result is perhaps not especially helpful. The typical usage is to provide the name of a *function*<sup>7</sup> you require help on. Perversely, `help` is a function and **MATLAB** provides help on `help`. The initial output to which is as follows:

```
» help help
help Display help text in Command Window.
```

In the course text, for each *function* that **MATLAB** provides a comprehensive help on, such as `help`, a simple summary version will be displayed in the right hand margin in a grey box.<sup>8</sup>

<sup>4</sup> i.e. a word, or sequence of characters that has a special meaning to **MATLAB** and it will act upon, as opposed to a sequence of characters that has not special meaning and **MATLAB** completely ignores.

<sup>5</sup> Otherwise known as Internet fishing.

<sup>6</sup> It is also possible to obtain context-specific help, e.g. on a specific (built-in) *function*, which we'll see in due course.

<sup>7</sup> Don't worry about what a *function* is yet.

<sup>8</sup> Refer to the section on 'How to use this Textbook'.

#### help

Typically takes a single parameter – the name of a *function*, and returns an entirely incomprehensible description of that function and its usage at the command line.

24 `str = 'do you like bananas?'`

## 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<sup>9</sup>. For instance – open up a blank worksheet in **Excel**, and in the very top left hand cell, enter the number 10. You can see visually, that **Excel** is referencing this location as column A, and row '1'. In fact, this location ('A1') is indicated in the Name Box to the left of the Formula Bar.

In **MATLAB**, a variable is associated a name to make things rather more easy and convenient. The name can be almost any sequence of characters you like, regardless of whether it is a real or fake word, as long as it does not contain numbers or special characters (e.g. #, \$, %, ...) or spaces. So actually, you are only left with continuous sequences of characters (otherwise known as 'words'). Note that you can create a variable name based on 2 (or more) real words, separated by an underscore (`_`) if that helps describe what the *variable* refers to. Valid variable names include:

```
A
B
cat
derpyhooves
this_is_boring_stuff
BIG
big10
```

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<sup>11</sup>.

So far so useless – you need to *assign* something to it. (The analogous situation is that when you first open an **Excel** spreadsheet and it is completely blank – you can still reference cell A1, but there is nothing in it.) Which brings us to quite 'what' and 'how'.

First of, you need to know that variables can have the following *types* of things assigned to them:

- **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 be allocated to storing it). This is something of a throw-back to the days when computers only had  $1/1000000$ <sup>th</sup> of the memory of your iPhone and were slower than half a lemon nailed

<sup>9</sup> In the bad old days, this pointer was the actual address in memory and might have looked something like `f04da105`.

<sup>10</sup> 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). I would strongly advise to stick to all lower case, or all upper case, to avoid possible future confusion. (or come up with a naming convention, of whatever sort (e.g. capital first letter), and stick to it.)

<sup>11</sup> 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.

to the floor. So we will not ion this text particularly worry about a numbers/computing concept called *precision*.

- **Real (floating point)**<sup>12</sup> – 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.

Be aware that you can configure<sup>13</sup> **MATLAB** to display a particular format for real numbers, e.g.

42.0

versus

4.2e+01

(there are identical *real* numbers, just a different display format).

- **String (character)** – One or more characters, but now allowing spaces (unlike in the case of naming *variables*).
- **Logical** – a *variable* that can be true or false<sup>14</sup> – we'll come to quite what this means later.
- **etc** – No, not a real *type*, but to note that **MATLAB** defines and recognises a whole bunch of other *variable types*, including **Complex** (**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 **MATLAB** documentation contains a full list (and/or go Internet Fishing).

To come back to **Excel** – if you select Format Cells (right-mouse-button-click over cell A1), you get to chose from a long list of 'formats', including Number and Text, and which have a loose correspondence with *types* in **MATLAB**.

The next thing to learn is ... to ideally, 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 some other programming languages, this should be avoided. However, even in **MATLAB**, *strings* and *reals* (or *integers*) are very different things.<sup>15</sup> 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*.<sup>16</sup> **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 to a *variable* using the *assignment operator* =<sup>17</sup>.

<sup>12</sup> The distinction (sort of) is that *floating point* is a specific representation of a **real** number.

<sup>13</sup> Under the menu item Preferences.

<sup>14</sup> As opposed to a Trump variable, that can have many different alternative states of 'true', although generally, a Trump 'true' is in fact 'false'. An entire new branch of mathematics and logical deduction has been created just to process al this.

<sup>15</sup> Again – in the **Excel** example, **Excel** will not let you add a **Number** and a **Text** value together, for instance. (Try it! You should see #VALUE! reported.)

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

Case #1 ( $\text{num2str}$ ) 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.

<sup>16</sup> Partially true. An Alternative Fact of sorts.

<sup>17</sup> This is NOT 'equals' in **MATLAB**. Or any sane programming language. We will see the *equality operator* shortly. = assigns the value or variable on its right, to the variable on the left.

```
26 str = 'do you like bananas?'
```

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 directly under the line you typed the command in at:

```
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 ...) i.e.

```
A = 10
```

is interpreted exactly the same as:

```
A=10
```

Pause ... this is sort of fundamental (to using **MATLAB**), what you have just done here. It is the equivalent of typing '10' into the cell A1 in Excel (assuming we can equate the **Excel** location A1 with the **MATLAB** variable A). In doing this, you have both: (a) created a variable A, and (b) assigned it a value of 10.

**MATLAB** will also report in the Workspace window, the name and value, *type* (unhelpfully 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<sup>18</sup>.

Pausing again ... if you want to remind yourself of the *variables* that you (or a program) have created – you can refer to the Workspace window.<sup>19</sup> Also listed here as noted above, is its value (and *type* etc). Another way to access the value of a *variable*, is to simply type in its name at the command line:

```
>> A
```

and **MATLAB** will parrot back:

```
A =  
10
```

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

<sup>18</sup> If you genuinely wanted an integer, there are ways to do this, such as using a type conversion function from *real* to *integer* (see above).

<sup>19</sup> There is a command line command for listing current variables (*whos*), but lets not bother with it.

```
B = apple
```

and **MATLAB** is far from happy. As it turns out, a sequence of characters can also refer to a *function*<sup>20</sup> in **MATLAB**, and this is what **MATLAB** looks for (i.e. a match to `apple` in the list of variable (and function) names). In other words, **MATLAB** does not know whether you intend `apple` to be a *string* or a *function*. It assumes *function* ... but cannot find one with that name and then gives up. To delineate `apple` as a *string*, you need to encase it in (single<sup>21</sup>) 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 the variable *Class* is now listed as `char`).<sup>22</sup>

Finally, it is possible to suppress output to the Command Window when making *variable assignments* – simply add a semi-colon (`;`) to the end of the *assignment* statement<sup>23</sup>, i.e.

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

Now, nothing is echoed back to the command line but the Workspace is still updated to reflect this *variable assignment*.

### 1.2.2 Numerical expressions and Arithmetic operators

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** —  $\wedge$  — 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).<sup>24</sup>
- **addition** —  $+$  — (guess).
- **subtraction** —  $-$  — again, obvious/intuitive.

Technically, these symbols are called (arithmetic) *operators*.

<sup>20</sup> 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.

<sup>21</sup> Double “ ” quotation marks will not work.

<sup>22</sup> Equally in **Excel**, you can simply type over a pre-existing value to replace it. e.g. you could type `banana` over the contents of cell A1 (that previous held the number 10).

<sup>23</sup> Again – your personal choice as to whether to include spaces or not between the `C`, `=`, `'banana'`, and `;` (Maybe try it both ways to convince yourself at least in this context, spaces do not matter.)

<sup>24</sup> 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).

28 str = 'do you like bananas?'

The order in which the arithmetic *operators* are written down is important and will execute them in a specific order (operators higher up the list, executed first), i.e. first ^, then \* and / (equally), and last, + and - (equally). There is also *negation*, when you change the sign of a *variable*, and which is executed immediately after exponentiation. e.g.

```
B = -A
```

The assignment operator (=)<sup>25</sup> comes last.

<sup>25</sup> This is NOT 'equals to'.

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 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 expect 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 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 – simply keep mind the existence of *relational and logical operators* and what they look like and we'll look into them some more later.

#### 1.2.4 Functions (built-in)

**MATLAB** provides numerous built-in *functions*<sup>26</sup>. These *functions* have specific names assigned to them, so care needs to be taken 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 text<sup>27</sup>. Common *functions* will be progressively introduced as this text progresses. 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). These 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  $\pi$ , is assigned to a built-in *function* 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  $\pi$ ) 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 ... and what exactly a *function* is, will all become apparent in due course ...)

#### 1.2.5 Miscellaneous commands

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

<sup>26</sup> We will be constructing our own later, at which point it should become apparent that there is nothing particularly special about them.

<sup>27</sup> A full list of functions can be found in the **MATLAB** Help Documentation under *functions*.

```
30 str = 'do you like bananas?'
```

- `clear` — Removes all variables from the workspace.
- `clear all` — (Removes all information from the workspace.)
- `close` — Closes the current figure window.
- `close all` — (Closes all figure windows.)
- `exit` — Exits **MATLAB** and hence enables an additional trip to Starbucks to be made.

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

For example – try assigning a value of 2.14159 to the variable `my_pie`. Having noted your mistake<sup>28</sup>, correct it. Do this by bring back the previous command, and editing the 2 to a 3 (and hit return). If you refer to the Workspace window, you can see that you have indeed successfully changed the value of `my_pie`.<sup>29</sup>

Note that there is also a Command History window that list all the previously issued commands and allows commands to be re-run by double-clicking on them. Copy-paste and re-running of single or multiple commands is also possible.

<sup>28</sup> An 'alternative' pi?

<sup>29</sup> The point is that this is much quicker than typing the entire line in again. Although later, when we start to put lines of code into files rather than typing everything at the command line, fixing mistakes becomes easier.

### 1.3 Vectors and arrays #1

So far, most of your *variables* have all be what are known as *scalars* – i.e. single numbers (*real* or *integer*)<sup>30</sup>. 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*<sup>31</sup> is the name commonly given to a 2-D array.)

#### 1.3.1 Creating vectors

Vectors are 1-D arrangements of numbers (or characters 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:

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

You might ponder on (or even try out) how you would create equivalent arrangements of numbers in an Excel sheet. From here on, it will rapidly become apparent why you would not want to be doing all this in **Excel**, although it remains a presumably familiar place to start from and makes links to the weirdness of **MATLAB** from.<sup>32</sup>

<sup>30</sup> An exception are when you assigned a string, which technically is a vector (assuming multiple characters in the string)

<sup>31</sup> Not to be confused with the film containing bad acting by Keanu Reeves.

The **colon operator** can be used to much more rapidly 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. **MATLAB** 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*

<sup>32</sup> As such, I encourage you to still think in **Excel** world as far as possible for a little while yet, because I think it will help get to grips with **MATLAB** array notation more quickly. And indeed, **MATLAB** has a very **Excel**-like array editor window to help bridge the gap.

32 str = 'do you like bananas?'

### 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*<sup>33</sup>.
2. Refer to the Workspace window, – initially, the contents of the vector are displayed (under column Value) and you have to count, but after a certain point, the size (and not contents) of the *vector* is displayed.
3. Use the `length` or `size` function (see Box).
4. Refer to the Workspace window ... but ... by default, the Size of variables is not one of the displayed columns (instead, it has to be added from Choose Columns right-mouse-button-click menu item)<sup>34</sup>.

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 the same orientation as C.<sup>35</sup> You can also use the *transpose operator* (see Box).

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

This next bit is maybe the single most important (and weird) part of **MATLAB**. As you go through this section (and also the later one on *matrices*) – have **Excel** open as a aid to visualize how **MATLAB** represents *arrays*. Start by entering the 5 numbers, from 0.5 to 2.5, in sequential cells, working down from A1 (this is the **MATLAB** vector B in the example that follows).

Values can be extracted (or read) from a vector by specifying the index (technically, this should be an *integer*, but **MATLAB** is pretty

<sup>33</sup> Try creating a *vector* from 1 to 100,000 and assign it to a *variable*. Refer to the use of the `color` operator (see earlier).

You will find that adding a semicolon to the end of the line to suppress output and instead viewing the vector in the Workspace Window.

<sup>34</sup> Although as per above – the size is displayed under Value for a sufficiently large *vector*.

#### 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).

<sup>35</sup> Note ... **MATLAB** gives the syntax as `.'`, whereas I always only ever added the `'` bit ... which works ...

#### 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 row vector. Simple! Lesson over.

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!)<sup>36</sup>

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).<sup>37</sup>

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:

```
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.)

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 `'`).

<sup>36</sup> Recognise the parallel with **Excel** here – the value in position 5 in the **MATLAB** vector *B*, is the same as specifying the contents of cell A5 in **Excel**.

<sup>37</sup> Again – e.g. in **Excel**, the sum of the 5 elements in column A (the equivalent 'vector'), would be =SUM(A1:A5).

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 to 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

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 (or similar) – this will form your *x-axis*, and the 2nd column ... whatever you like.<sup>38</sup> 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`.

With any new **MATLAB** command (*function*), get into the habit of looking up the help text (also refer to alternative/simplified help provided in this text). The key information that will get you started appears at the very top of the text that help returns on `plot`:

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

This tells you that you need to pass to `plot`, 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".<sup>39</sup>

This ... is easily the least professional plot ever (aside from anything created in **Excel**). And one that breaks all the most basic rules of scientific presentation, such as an absence of any labelling of 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. This is the default output 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 two data vectors. In the

<sup>38</sup> 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.

<sup>39</sup> If you cannot see the figure window ... check that the window is not hidden behind the main **MATLAB** program window!

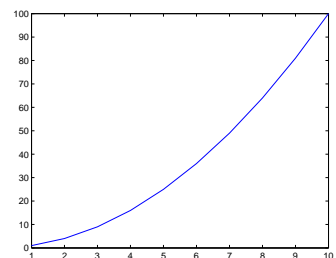


Figure 1.1: Example of the default output of the `plot` function.

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 relatively simply/whole numbers. If however the maximum  $y$  value was very 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).<sup>40</sup> 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).

### 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 **MATLAB** 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 text 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.

Note that in the usage of all the above listed commands, they all require something to be passed within a set of parentheses – ( ). In fact, they are all **MATLAB** functions and require an input (hence the use of the parentheses). Some of the functions require a *string* input,

<sup>40</sup> 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.

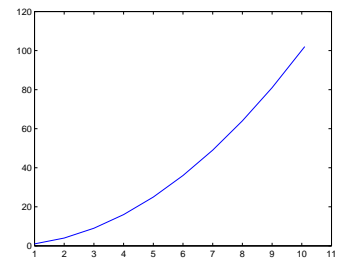


Figure 1.2: A plot illustrating axis auto-scaling (maximum  $x$  and  $y$  values now slightly larger than 10 and 100, respectively).

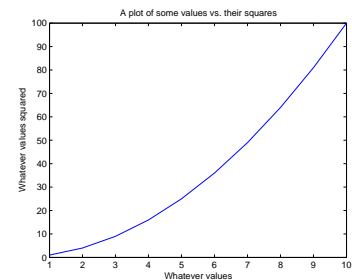


Figure 1.3: A (only very slightly) improved plot.

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');

```



36 str = 'do you like bananas?'

such as the name of the title in `title`, and this must be encased in quotation marks – ' ' to designate it a string rather than a *variable* name.<sup>41</sup>

### 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  $2\pi$ , 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<sup>42</sup>. 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.

### 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*<sup>43</sup>. 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).

<sup>41</sup> You could instead assign a *string* to a *variable*, and then pass the *variable* name (no quotation marks).

#### 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  $ymin$ ,  $ymax$ ).

<sup>42</sup> » help subplot

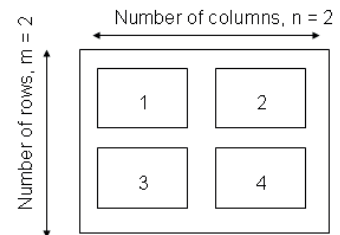


Figure 1.4: Arrangement of subplots.

<sup>43</sup> 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.
```



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

To close the current (active) Figure window, the command is:

```
> close
```

To close all currently open Figure windows:

```
> close all
```

```
38 str = 'do you like bananas?'
```

## 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<sup>44</sup>:

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

In math-speak, this would be equivalent to:

$$A = \begin{pmatrix} 15 & 7 & 11 & 6 \\ 13 & 1 & 6 & 10 \\ 21 & 17 & 5 & 3 \\ 5 & 15 & 20 & 9 \end{pmatrix}$$

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 **Excel** 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 this class because that will give you far more power and flexibility in applying numerical techniques to problem solving, and will form the basis

<sup>44</sup> Remember that you can add an `;` to the end of the line to prevent the results of the *variable* assignment being displayed in the Command Window.

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).

Finally – a fundamental way of accessing data that you need to learn and be familiar with, is to employ the *colon 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)`<sup>45</sup>, 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!)

### 1.5.2 Basic matrix manipulation

You can treat *vectors* and *matrixes* (or parts of *vectors* and *matrixes*), 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 *arithmetic operators*<sup>46</sup>. For example, for *vector* *B* in the earlier example,

```
» 2*B
ans =
```

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.
```

<sup>45</sup> Remembering the HUGE hint above in 100 pt font as to the order of rows and columns ...

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).

<sup>46</sup> 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.)

```
40 str = 'do you like bananas?'
```

```
0 2 3 4 5
```

and

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

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.) The **MATLAB** function for this is `sum`.

QUESTION: What is the sum of the 2nd row of C? For a *matrix* (rather than a *vector*) as input, `sum` returns the individual 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)<sup>47</sup>.

```
> 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
```

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.'

<sup>47</sup>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:

`flipud` – 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 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
```

to give you a row *vector* format that corresponds to the rows of the original *matrix* C.<sup>48</sup>

Finally, if you wanted the sum of *\*all\** the elements in the *matrix* C in the example above, you could sum all the columns to give you a row *vector* of partial sums, and then sum the elements in the row *vector* to give you the grand total sum of all the elements. You can do this, either in completely separate steps<sup>49</sup>:

```
» D = sum(C);
» E = sum(D);
```

or all in one go:

```
» F = sum(sum(C));
```

It does not matter if you sum the column of C first, or the row first – maybe test this to satisfy yourself that this is true.

### 1.5.3 Some matrix math :(

We will not concern ourselves overly with multiplying *vectors* and *matrices* together ... but you should be aware that **MATLAB** can do matrix math. For now, it is worth noting the difference between `*` and `.*` operators in the context of arrays. For example, consider 2 vectors, A and B:

```
» A = [1 1 2 2];
» B = [1 2 3 4];
```

To multiply the elements of A and B together pair-wise, use `.*`:

```
» C = A.*B
C =
    1 2 6 8
```

Without the dot, you get the vector product ... well, you would if the *vectors* were in an appropriate orientation, i.e.:

<sup>48</sup> 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.

<sup>49</sup> In general in programming – use as many smaller, separate steps as you like and are most comfortable. The more you break down the calculation, the clearer it will be to you and the easier to debug if things go wrong. However, this does come at the expense of longer and longer code and sometimes more compact code is easier to deal with.

```
42 str = 'do you like bananas?'
```

$$\begin{pmatrix} 1 & 1 & 2 & 2 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

which you get by typing:

```
> C = A*B'
C =
    17
```

(which is calculated from:  $1 \times 1 + 1 \times 2 + 2 \times 3 + 2 \times 4$ ).

An example of the equivalent matrix usage is:

```
> D = [1 1; 2 2];
> E = [1 2; 3 4];
```

The pair-wise multiplication of each element of the 1st matrix with the corresponding element of the 2nd matrix is:

```
> F = E.*E
F =
    1  4
    9 16
```

In contrast, for matrix multiplication, written in math-speak as:

$$\begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

we would write:

```
> F = E*E
F =
    7 10
   15 22
```

If your matrix math is rusty and you are not following this, maybe refresh it (your memory of basic matrix math).

## 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 accidentally 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 the file you want to load in is not in the directory **MATLAB** is 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.

**MATLAB** has a default directory that it starts up in and looks at first. For basic Windoz installations<sup>50</sup> of the software, this directory is:

```
C:\Users\mushroom\Documents\MATLAB
```

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.

It is not necessarily to have all your files end up here, so there is a way to change the **MATLAB** directory that you are working in which work in a similar way to **UNIX/LINUX** for those of you who are familiar with navigating your way around these operating systems. You can change the directory that **MATLAB** is working from by typing:

```
» cd DIRECTORY_PATH
```

where `DIRECTORY_PATH` is the path to the directory in which you want to work from and where you want your data files (and later, code files) to live. For example, if (in **Windoz**) you have mounted a USB drive, it might be assigned drive letter E:. To change **MATLAB**'s working directory to this drive, you would type:

```
» cd E:
```

If, you have a directory working on the USB drive, you could change **MATLAB**'s working directory by:

```
» cd E:\working
```

Note that an alternative format (syntax) for `cd` is as a *function*, e.g.,

<sup>50</sup> At installation, this directory can be specified and hence may not be this one. Also – different operating systems will have different default locations.

#### `load`

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

```
» load(FILENAME)
```

where `FILENAME` is the name of the file (remember: `FILENAME` needs to be a string and enclosed in quotation marks, OR, a variable that points to a string.

The file might be plain text (ASCII) or a **MATLAB** workspace file (see below). 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 can instead choose to assign the imported data to a variable name of your choice, by e.g.:

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

(all one line)

```
44 str = 'do you like bananas?'
```

```
» cd('E:\working')
```

with the directory path passed as a string.

Another alternative is to add a 'search path' (`addpath`) so that **MATLAB** knows of an additional place to look for files. For example:

```
» addpath('E:\working')
```

would keep your current working directory unchanged, but tell **MATLAB** to also look in the directory `E:\` for files.

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. The *GUI* can also be used to change the directory you are working from (duplicating the functionality of the `cd` command) and add paths to search (duplicating the functionality of the `addpath` command).

In summary: go with whatever works best for you in terms of working directory. Easiest might be simply to work directly from a directory on your USB pen-drive (and e.g. have a named different directory for each week of class).

### 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)<sup>51</sup>.

As a brief exercise and practice using `load` – first download the data file `etheridge_etal_1996.txt` from the course webpage<sup>52</sup>. You might start by viewing the contents of the file by opening it in any text viewer (or **Excel**)<sup>53</sup>. 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).

Now, import the data into the **MATLAB** workspace using the `load` command. Because the data is a plain text (ASCII) format and not a special **MATLAB** `.mat` file, you need to specify the format as `'ascii'` when using the `load` command (see box or help):

```
» load('etheridge_etal_1996.txt', '-ascii');
```

If you tell **MATLAB** nothing different, it will create a variable for you containing the file content, with the variable name based on the filename (minus the extension). If you prefer a different variable name, then simply pass the results of the `load` command – the contents of your file – to a different *variable*, e.g.

#### `addpath`

The command `addpath` will add a search path to the **MATLAB** workspace. The syntax is:

```
addpath(DIRECTORY_PATH)
where DIRECTORY_PATH is
a string (characters in between
inverted commas) or name of a
variable containing a string.
```

<sup>51</sup> 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.

<sup>52</sup> <http://www.seao2.info/teaching.html>

<sup>53</sup> In fact, you could even try first plotting it in **Excel**.



```
» mydata = load('etheridge_etal_1996.txt','-ascii');
```

instead assigns the load-ed data to the *variable* mydata.

Try typing the name of the *variable* that was automatically created (etheridge\_etal\_1996) (or the one you chose if you assigned the imported data to a specific variable name as per detailed in the Box) to provide a crude view of the data. To view the contents of the *variable* in the Variables window – 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.

For practice, try plotting the data<sup>54</sup> and remembering to label the figure appropriately<sup>55</sup>. However ... remember, the format of the **MATLAB** plot function is:

```
plot(X,Y) plots vector Y versus vector X
```

So you need to specify each column of the data (i.e. each *vector*) separately and explicitly.<sup>56</sup>

You can do this step-by-step, and create yourself 2 vectors, one for the x-values and one for the y-values, and then plot:

```
» X = etheridge_etal_1996(:,1);
» Y = etheridge_etal_1996(:,2);
» plot(X,Y);
```

Or go straight for the kill:

```
» plot(etheridge_etal_1996(:,1),etheridge_etal_1996(:,2));
```

Breaking things down like is an equally valid way of doing things. It is longer ... taking 3 lines rather than 1, but the most important thing is to be happy that you understand what is going on. If breaking things down into multiple lines and creating new *variables* helps – DO IT! Ultimately, you should end up with something like Figure 1.6.

### 1.6.3 Saving and exporting data

Specific variables can be saved in a plain text (ASCII format) by means of the save function (and then re-loaded in using load). You have to specify that you want a text format (rather than the default **MATLAB** .mat workspace format) – see Box. Try re-saving the ice-core data as an ASCII format text file (with a new filename) ... and then load it in again.

### 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

<sup>54</sup> using plot

<sup>55</sup> FYI: the first column of the data and x-axis is year, and the 2nd column of the data and y-axis is the mixing ratio of CO<sub>2</sub> in air in units of ppm.

<sup>56</sup> If you just type plot and pass the (here: default) name of the data array:

```
» plot(etheridge_etal_1996);
```

... strange ... things are happening (as per Figure 1.5). In fact, **MATLAB** is doing what **Excel** would in a Line Chart with 2 columns of data selected – rather than plot y (2nd column) vs. x (1st column), the values of both columns are plotted against row number. Which is why you should remember to use the Scatter (or (X,Y) Chart in **Excel** for plotting (x,y) data.

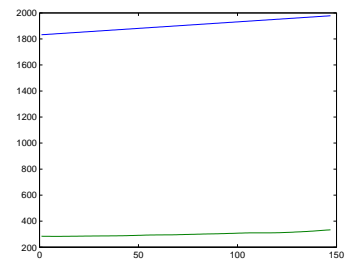


Figure 1.5: Result of simply throwing the entire data matrix at plot ...

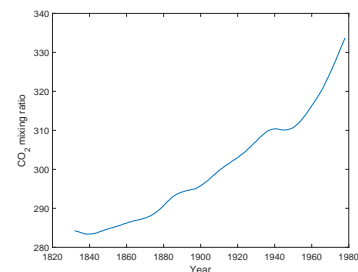


Figure 1.6: Spline fit to measured changes in CO<sub>2</sub> concentration in Law Dome ice core, following Etheridge et al. [1996].

46 `str = 'do you like bananas?'`

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. Try saving the current Workspace, then close down the **MATLAB** program. Re-running it, and then loading in your saved `.mat` file.<sup>57</sup>

Hopefully ... all your loaded/created variables etc. have been recovered ... ?

<sup>57</sup>This sequence is going to look something like:

```
» save mystuff
» exit
```

...

```
load mystuff
```

Remember that when you re-start **MATLAB** you may have to change directories, add a path (`addpath`), or provide a full path to the `.mat` file, depending on where you saved it.

#### **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.

**MATLAB's** proprietary file format for saving the contents of your current Workspace is indicated by a `.mat` file name extension (in Windoz).

## 1.7 Basic data processing (and yet more plotting)

This section runs through a couple of common basic data manipulation/processes techniques follow, plus some further plotting/visualization.

### 1.7.1 Sorting data (in arrays)

As an example to kick-off some data-processing tricks, load in the dataset of ('proxy') reconstructed atmospheric CO<sub>2</sub> concentrations spanning the Phanerozoic: `paleo_CO2_data.txt`. You can just import it into **MATLAB** using the `load` function – remember the specific syntax of `load`:

```
» load(FILENAME)
```

where your `FILENAME` is `paleo_CO2_data.txt` and needs to be passed to `load` as a string, i.e.

```
» load('paleo_CO2_data.txt')
```

If you tell **MATLAB** nothing different (and do not assign the results of the `load` function to a different *variable* name), **MATLAB** will create a *variable* called `paleo_CO2_data` (see Workspace).

If you view the contents of the variable `paleo_CO2_data` (or whatever you might have passed the `load`-ed data to), you will see that there is a slight complication – unlike the ice core CO<sub>2</sub> dataset, you now have 4 columns in this array<sup>58</sup>. The first column is age (Ma), the second the mean CO<sub>2</sub> value, while the 3d and 4th columns are the low and high, respectively, uncertainty limits of the CO<sub>2</sub> value.

Recalling how to reference specific columns of data in a matrix<sup>59</sup>, and either referencing the columns of the array directly, or creating yourself separate vectors `X` and `Y` (see earlier) – plot the mean paleo CO<sub>2</sub> 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 1.7.

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 CO<sub>2</sub> 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<sub>2</sub> data, this is not the case and time (age in Myr) does not progress monotonically in an always-getting-older (or always-getting younger) fashion.<sup>60</sup>

<sup>58</sup> Remember that you can diagnose its size with ... `size` (or refer to the Workspace window)

<sup>59</sup> HINT: the colon operator (see earlier).

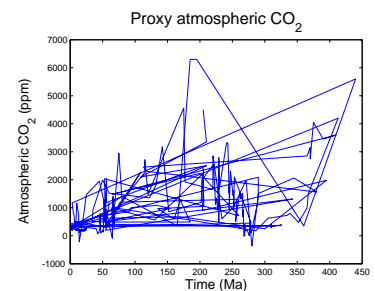


Figure 1.7: proxy reconstructed past variability in atmospheric CO<sub>2</sub>.

<sup>60</sup> In fact, in the original, full version of the data, ordering is by proxy type first, and then study citation, and only then age ...

48 str = 'do you like bananas?'

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
```

However, 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. So this is not the *function* that you are looking for.

This is where it is worth paying attention to the bottom of **MATLAB** help and the `see also` section. In this case, **MATLAB** lists `sortrows` as a possibility. The help text on this looks a little more promising. It is still slightly opaque (so also see `Box`), so the best thing to do is to try it (and view the results)!

```
» ordered_data = sortrows(paleo_CO2_data)
```

where the result of sorting the rows (of all columns) I have assigned to the variable `ordered_data`. If you now try plotting this, e.g.

```
» plot(sorted_data(:,1),sorted_data(:,2));
```

it looks rather better – Figure 1.8. (This is a good illustration of a guess of a *function* that was 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  $\text{CO}_2$  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).

A little later you will meet an alternative plotting *function* that does not require the data to be sorted into any sort of order. But you should note that you can also use `plot`, but omitting the line segments by specifying only a symbol, e.g.

```
plot(x,y,'ro');
```

(here, plotting circles for the data points in red) so that it does not matter in which order the individual points are plotted (and the same result is obtain from both sorted and un-sorted data).

#### `sortrows`

In its simplest usage:

```
» B = sortrows(A)
```

... "sorts the rows of a matrix in ascending order based on the elements in the first column. When the first column contains repeated elements, `sortrows` sorts according to the next column and repeats this behavior for succeeding equal values."

So, if the first column of the matrix was time, the data would be sorted into ascending time.

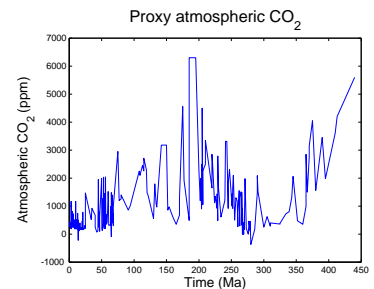


Figure 1.8: Proxy reconstructed past variability in atmospheric  $\text{CO}_2$  (sorted data).

### 1.7.2 Data scaling

As an example practicing some basic data scaling: download the historical global temperature anomaly dataset<sup>61</sup>:

temperature\_globalanom.txt

The columns are: (1) year , (2) annual mean ocean+land surface temperature anomaly (i.e. temperature change relate to some reference value, where here is the observed 1901-2000 mean). Remember, you can load and assign data to an easier-to remember variable by e.g.:

```
» data1 = load('temperature_globalanom.txt', '-ascii');
```

Plot the annual mean temperature anomaly for the full range of years, as per Figure 1.9. (plus labels, title, etc etc).

The 20th century average global temperature across land and ocean surface areas is apparently 13.9°C. So first, change the temperature anomaly data into absolute temperatures – you'll do this by by adding the 20th century global average value, 13.9, to all the data values (the second column of your array).<sup>62</sup>

Now re-plot.

Next, convert the temperature units from °C to °F. An approximate conversion is:

$$T_{(^{\circ}\text{F})} = 1.8 \times T_{(^{\circ}\text{C})} + 32$$

where  $T_{(^{\circ}\text{F})}$  is the (new) temperature in °F, and  $T_{(^{\circ}\text{C})}$  the (old) temperature in °C.

For this, you will need to take your data (which is the 2nd column of the array), e.g. `data1(:,2)`, multiple by 1.8 as per the equation ( $1.8 * \text{data1}(:,2)$ ) and then add 32 ( $1.8 * \text{data1}(:,2) + 32$ )<sup>63</sup>. And ... assign the results of this to a new vector or array. Or you can replace the original column (if you are feeling confident), e.g.

```
» data1(:,2) = 1.8*data1(:,2) + 32;
```

The aim is to obtain a data array in **MATLAB**, with year as the first column (year, as per the original data) and the 2nd column as annual mean temperature in units of °F.

If it helps – play the data conversion game in **Excel** first (e.g. creating new columns in a spreadsheet to firstly hold absolute temperatures rather than anomalies, and then temperatures in °F rather than °C). Also if it helps – create a new array with the modified temperature units data in (rather than replacing the 2nd column of the original array, `data1`). You can also do the conversion in 2 stages – multiplying the (absolute) temperature (°C) by 1.8 first (perhaps creating a new array to hold this in), then adding 32.

<sup>61</sup> NOAA

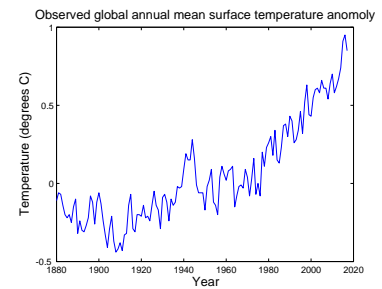


Figure 1.9: Observed annual global mean surface temperature anomaly (compared to year 1910 to 2000 average).

<sup>62</sup> Remember – you can increase the value of every element in an array, by simply adding that number, e.g. if `A` is you array of data, and `B` is a *scalar* (the value you want to increase all array values by);

```
» C = A + B;
```

will have the effect of adding `B` to ever element in `A`, and assigning to a new array, `C`. Or alternatively, you can replace the contents of array `A` with the new values:

```
» A = A + B;
```

In your specific example:

```
» data1(:,2) = ...
   data1(:,2) + 13.9
```

(all one line) will have the effect of taking the 2nd column of the array `data1`, adding 13.9 to all the values, and writing the new values back into the 2nd column of the array `data1`.

<sup>63</sup> If you have any doubts as to the order in which the operators are applied, add a set of parentheses, e.g.

```
(1.8*data1(:,2)) + 32
```

```
50 str = 'do you like bananas?'
```

Re-plot (in **MATLAB**) once again the final temperature trends in °F. This should look like Figure 1.10.

### 1.7.3 Data (row) deletion

For some practice in data filtering (row deletion) and a little further data manipulation and graphing: download the historical temperature dataset for Riverside<sup>64</sup>:

```
temperature_riverside.txt
```

If you view the data file in a text editor or import into **Excel**, you can read the column headers and find out what the different columns of data are (I am not telling you!). Note that the (1st) line of the file containing the column header labels starts with a % symbol (telling **MATLAB** to ignore this line and not attempt to read in the 'data' on it). Again, you might assign the read-in data to a different variable name, e.g.

```
» data2 = load('temperature_globalanom.txt', '-ascii');
```

First create a plot (with appropriate labels) of this data. You want the annual mean temperature – the last (14th) column, plotted vs. year (the first column), so:

```
» plot(data2(:,1),data2(:,14));
```

or:

```
» plot(data2(:,1),data2(:,end));
```

It would be 'nice' ... to make some direct comparison between the observed global temperature increase and that occurring in Riverside, e.g. to help answer questions such as 'Are temperatures increasing faster in Riverside than the global mean?', and hence 'Will global warming impacts likely be worse or less severe in the Riverside area as compared globally?'. To do this, we need both data sets – global and for the Riverside area – to be on a comparable scale.

You could certainly simply plot both global mean and Riverside annual mean temperatures alongside each other, using the same units, e.g. °F as you have previously converted the global mean temperatures to °F, which is the same units as the Riverside temperature data. You could have 2 sperate plots and visually compare them, but this is not very clever nor necessarily useful in making any sort of quantitative comparison. For instance – contrast the global data (re-scaled to absolute degrees F) in Figure 1.10, with the Riverside data, in Figure 1.11.<sup>65</sup>

There are two main problems<sup>66</sup> in making a useful comparison –

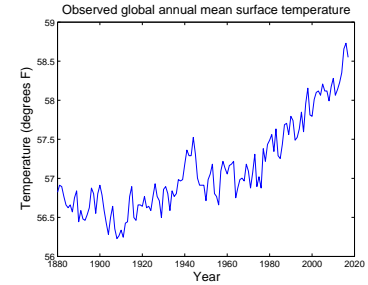


Figure 1.10: Observed annual global mean surface temperature.

<sup>64</sup> NOAA

#### comment symbol

% – is a special symbol that when **MATLAB** sees it, it ignore the entire line. This is known as a comment symbol (of identifier) and allows you to have lines of comments in amongst the lines of code.

Equivalently, when **MATLAB** loads in a ASCII data file, any line in which the % symbol appears, **MATLAB** ignores and does not load in. Hence, column header descriptions (or any other file description information) can be included in the file as long as the line starts with a %.

<sup>65</sup> There are also some odd-looking artifacts ('spikes') in the raw data that we will want to deal with in some way.

<sup>66</sup> Plus artifacts in the raw data.

firstly, the two data sets are on different y-axis scales (but luckily on the same x-axis, year scale), with the global data temperature scale going from 56 to 59°F, and the local, Riverside temperature data scale going from 50 to 85°F.

The limits can be specified and made common between the 2 plots using the `axis` command that you saw earlier. You could, for instance, not worry about truncating the spurious spikes in the Riverside temperature data and set the y-axis limits for both plots to e.g. 55 to 70°F. (You are still left with comparing across 2 different plots, which we will fix in a subsequent section by means of the command `hold on`.) However, there is still an inconvenient offset between the global mean temperature and that at Riverside.

Recall that the original global temperature data was given as an anomaly compared to the average over some baseline (or reference) period – in this case, year 1910 to 2000. If we treated both data sets the same, and transformed the Riverside temperature data into a comparable anomaly, direct comparison could be made. To create an anomaly of the Riverside temperature data, relative to the mean of the data for years 1910 to 2000, requires the mean of the years 1910 to 2000 ... this is not difficult to do, but it better left for another time ... For now, take it as having a value of 64.2°F. So to create an anomaly for the Riverside temperature data, simply subtract 64.2 from the values in the annual mean data column (the 14th column, which you can also specify with `end`).

Finally, go back to the original global mean anomaly values (re-load the data set if necessary) and convert from the anomaly in °C to °F (i.e. simply multiply by 1.8 – no offset (32°F) is required in this particular case). If you additionally, chose and set a sensible common y-axis scale for both plots, you might end up with a pair of graphs looking like Figures 1.12 and 1.13.<sup>67</sup>

Finally to some data filtering (row deletion) – it was mentioned earlier that there were potential 'artifacts' in the Riverside mean annual temperature data. If you view the loaded in data array in the Variable viewing window (double-click on the `temperature_riverside` variable name in the Workspace window), you can see for a number of years and months, rather than numbers, 'NaN's in the cells. NaN stands for 'Not a Number' and indicates that there is no (valid) numerical value for that array position (cell). The impact of there being a number of months of data missing, is that the annual mean is no longer a true annual mean but rather simply the mean of whatever monthly data exists for any particular year. For example, year 2008 has no data other than during the summer and the annual mean is hence simply

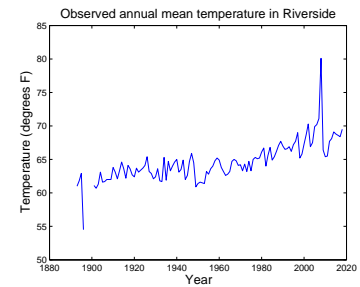


Figure 1.11: Observed annual mean surface temperature in Riverside.

<sup>67</sup> There are all sorts of likely reasons for the differences. Firstly, the global mean surface temperature rise includes both ocean surface and land surface. Because of the higher heat capacity of the ocean, the ocean surface warms slower than the land, and the ocean accounts for ca. 70% of the total global surface area. So it is somewhat inevitable that the warming trend will be stronger in Riverside. It may also be that the Riverside data is influenced by the 'urban heat island' effect, in which longo-term measured trends can be affected by increasing urbanization of the area surrounding the weather station. It may also be that the latitude and specific location of Riverside, sees much more warming than the global mean.

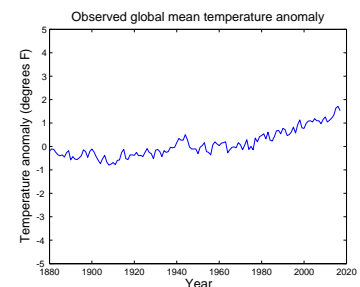


Figure 1.12: Observed global annual mean surface temperature anomaly, relative to the mean of 1910 through 2000.



52 str = 'do you like bananas?'

equal to the July temperatures!

One could address this by removing the years with (substantially) incomplete monthly data from the data file<sup>68</sup> and prior to loading into **MATLAB**. Or would could process the data once in **MATLAB**. This can be done by assigning to particular row (vector) of data, an empty vector ([ ]).

Taking first a simple example of a column vector:

$$A = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

from which we wish to remove the 2nd row. In **MATLAB** we would create the vector by:

```
» A = [1; 2; 3];
```

and then remove the 2nd row by setting it to an empty element:

```
» A(2) = [];
```

Similarly, to remove the 2nd row of:

$$B = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

```
» B = [1, 4; 2, 5; 3, 6];
```

```
» B(2,:) = [];
```

(instead removing e.g. the 1st column would be `B(:,1) = []`)

So back to the temperature data – to for example remove the row containing the year 2008 data<sup>69</sup>, which is row 11, we write:

```
» temperature_riverside(11,:)=[];
```

Play this 'game' – deleting as many row as you think result in biased means (because of missing monthly data)<sup>70</sup>, with the Riverside temperature data, and re-plot the results. For example, the result of removing ALL the rows with missing monthly data<sup>71</sup>, results in Figure 1.14.

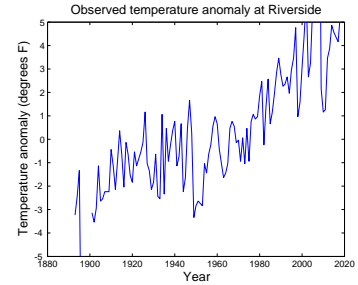


Figure 1.13: Observed annual mean surface temperature anomaly, relative to the mean of 1910 through 2000, at Riverside.

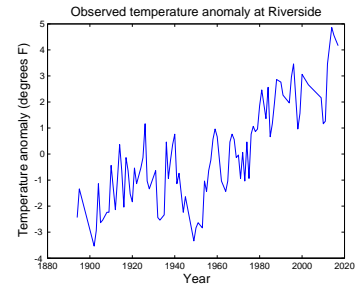


Figure 1.14: Observed annual mean surface temperature anomaly, relative to the mean of 1910 through 2000, at Riverside, filtered to remove years with missing monthly data.

<sup>68</sup> i.e. simply deleting the line in the file.

<sup>69</sup> You can also delete rows (and columns) if you open up the **MATLAB** Array window (double-click on the variable name in the Workspace window). And ... edit/replace values ...

<sup>70</sup> (being aware that as you delete rows, the numbering of the subsequent rows changes as the array size shrinks ...)

<sup>71</sup> (There are simple and quick ways of doing this in **MATLAB** that we will see later.)



## 1.8 Nicer 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`, is 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 1.15. A summary of a few of the more common plotting options is provided in the Box.

### 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<sub>2</sub> 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**? (There is a function for this – `errorbar`.), 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 plot with different symbols, with the min and max values as points. Or plotted min and max lines as a bounding envelope (lines). All of these require some further little trick in **MATLAB**, which involves the command `hold`. This is nice and simple and takes the additional (2nd) word: `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.

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.

To use them, add a new parameter when you call the `plot` function – whereas before, you typed, for plotting a vector  $y$  against  $x$ :

```
plot(x,y);
```

now add, following a comma, the point and line style option, encased in a pair of inverted commas, e.g. for a red, dashed line:

```
plot(x,y, 'r-');
```

and for blue circles (no line):

```
plot(x,y, 'bo');
```

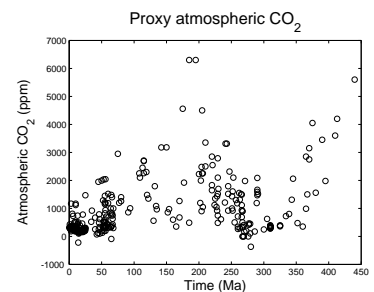


Figure 1.15: Proxy reconstructed past variability in atmospheric CO<sub>2</sub> (sorted data).

AS AN EXAMPLE – set:

```
» hold on
```

and then plot the minimum and maximum CO<sub>2</sub> values (columns #3 and #4) in different symbols and different colors, on top of your existing plot. If you want to then label what the different lines or sets of points are, you can add a legend with the `legend` command. For instance, if you have managed to successfully plot the mean CO<sub>2</sub> values as discrete black circles, and the minimum and maximum uncertainty limits as blue and red lines, respectively, you could type:

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

(all one line) and it should end up looking like Figure 1.16.

---

Returning to the previous plotting Example, of the observed historical trend in global mean temperature vs. the temperature recorded in Riverside – you now know how to plot BOTH temperature anomaly trends in the same figure window (and the same panel), so try it!

### 1.8.3 Changing label font size (and type)

The axis and title labels, by default, can be difficult to read when the graphics are saved and then imported into a document/paper. You can change the size of text as you create axis captions and figure titles etc., by specifying the value of an additional (text size) parameter in the function. For example, to increase the size of the x-axis label to a 14pt font:

```
» xlabel('Year','FontSize',14);
```

Here – after the you have passed the string you wish to appear to the **MATLAB** `xlabel` function ('Year'), there is a pair of additional parameters:

```
'FontSize',14
```

The first additional parameter specifies the aspect of the axis label that you wish to change (here: 'FontSize'), and the 2nd parameter of the pair, is the (new) value (here: 14).<sup>72</sup> Similarly, the y-axis label and title text size can be adjusted in exactly the same way.

Other property parameters<sup>73</sup> that might be useful (to change), are (with example changes):

```
'FontName','Courier'
'FontWeight','bold'
'FontAngle','italic'
```

#### errorbar

Works like `plot`, except it adds error bars. The 2 most useful usages are:

```
errorbar(x,y,err) plots y
versus x and draws a vertical error
bar at each data point. All of x, y,
and err, are vectors (all of the same
length).
```

```
errorbar(x,y,neg,pos) draws
a vertical error bar at each data
point, where neg determines the
length below the data point and pos
determines the length above the data
point, respectively. All of x, y, neg,
and pos, are vectors (all of the same
length).
```

#### hold

According to **MATLAB** help:

`hold on` – retains plots in the current axes so that new plots added to the axes do not delete existing plots.

`hold off` sets the hold state to off so that new plots added to the axes clear existing plots and reset all axes properties.

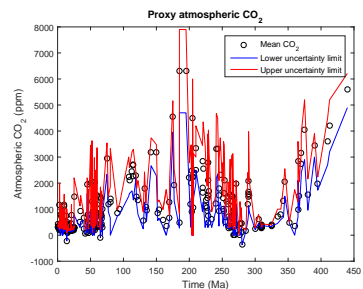


Figure 1.16: Proxy reconstructed past variability in atmospheric CO<sub>2</sub> (sorted data).

<sup>72</sup> See help on `xlabel`.

<sup>73</sup> Again – refer to **MATLAB** help.

### 1.8.4 Scatter plots

Returning back to the Phanerozoic proxy (CO<sub>2</sub>) data, we can now put a different (graphical) spin on it.

Consider ... `scatter`. In fact, don't just considered it, help on it (`>> help scatter`). 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<sup>74</sup>, and `scatter`-plot the (mean) CO<sub>2</sub> data.

Perhaps a little disappointingly, the default (Figure 1.17) (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<sub>2</sub> value. e.g.

```
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):

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

resulting in Figure 1.18.

### 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 data matrix: `model_grid.txt`. Then, 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). Lastly, go ahead and employ the `pcolor` function in its simplest possible usage (see Box) to visualize the data. You can see (Figure 1.19) that it is ... something. Maybe a little like the continents, but up-side-down at the very least. What to do?

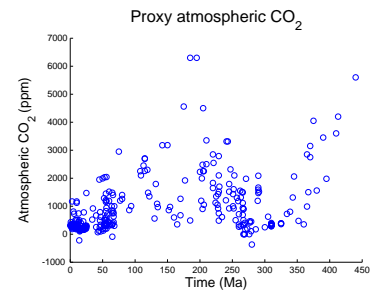


Figure 1.17: Proxy reconstructed past variability in atmospheric CO<sub>2</sub> (scatter plot).

<sup>74</sup> See earlier.

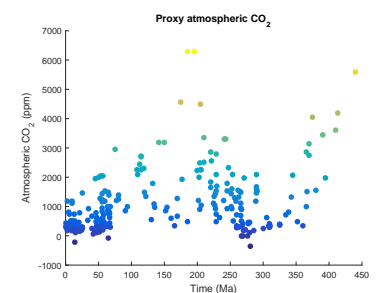


Figure 1.18: Proxy reconstructed past variability in atmospheric CO<sub>2</sub> (scatter plot).

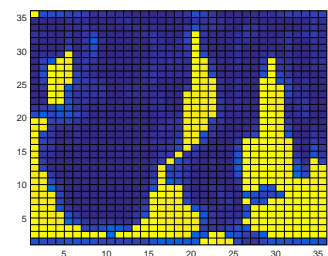


Figure 1.19: A 2D plot of some random gridded model data.

Well, it is a good job that you remember how to re-orientate arrays, right?<sup>75</sup> If you guess right first time (three different basic transformations of a matrix were described), you will get Figure 1.20.

Next try something very similar. but using the `image` function.<sup>76</sup>

What is the point of this? You now have the ability to simply visualise a gridded dataset. Later, we'll be doing it more formally 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.

<sup>75</sup> You don't? See earlier in the Chapter ...

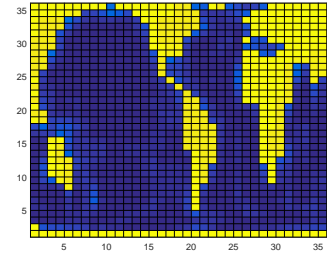


Figure 1.20: A 2D plot of some random gridded model data ... but with the underlying data matrix re-orientated before plotting.

<sup>76</sup> 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.

#### `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.

### 1.9 Further matrix math (systems of equations)

You can also use **MATLAB**'s powerful matrix functionality to solve real-world problems for you.

AS AN EXAMPLE – consider the Great Lakes – the largest lake system in the world. They have on their shores some of the greatest cities ... as well as some of North America's worst hockey teams. More importantly, much of the region is heavily industrialised and there is hence an exciting potential for pollution input into the lakes and hence a contrived numerical modelling exercise.

The layout of the lake system is shown schematically in Figure 1.21, together with the mean volumes of the lakes and the annual flow rate of water out of them.

A cocktail of heavy metals pours into each lake, the amount dependent largely upon the population within the catchments of the lake. The input rates to each of the 5 lakes are given below.

Lake	Heavy Metal Input ( $\text{kg yr}^{-1}$ )
Superior	$1.0 \times 10^3$
Michigan	$4.5 \times 10^3$
Huron	$1.0 \times 10^3$
Erie	$3.5 \times 10^3$
Ontario	$3.0 \times 10^3$

The steady state concentration of heavy metals in the Great Lake system (the steady state solution being the state in which none of the concentrations in any of the lakes is changing) is something that you can find an analytical solution for. You have 5 unknowns (the concentration in each of the 5 lakes) and you can write down a series of 5 equations involving these unknowns. (There is slightly more to it than this, as there must also exist an inverse for the matrix, which is not always the case ...)

Lets call the concentrations ( $\text{kg km}^{-3}$ ) of heavy metals in the lakes;  $c_S$ ,  $c_M$ ,  $c_H$ ,  $c_E$ , and  $c_O$  (for; Superior, Michigan, Huron, Erie, and Ontario, respectively). At steady-state, the inputs of heavy metals must exactly balance the outputs from each lake (otherwise, the concentration in the lake would change and the system would not be at steady-state). We can write a series of mass-balance equations for the 5 lakes. For instance, in Lake Superior, the metal input flux is  $1.0 \times 10^3 \text{ kg yr}^{-1}$  ( $1000 \text{ kg yr}^{-1}$ ). This must balance the loss of metals in the river outflow if the concentration of metals in the lake is to remain constant. The water outflow rate that is given to you is  $63 \text{ km}^3 \text{ yr}^{-1}$ . The metal outflow flux is then just the concentration of metals in the water ( $c_S$ ), times by the water flow;  $63 * c_S$ . Thus, for

Table 1.1: Pollution input rates to each of the 5 lakes.

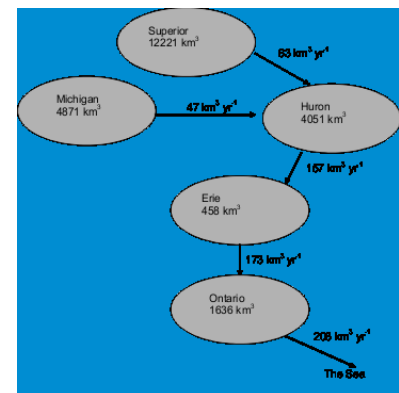


Figure 1.21: Lake volumes and river flow rates in the Great Lakes system.

58 str = 'do you like bananas?'

Lake Superior, we can write  $1000 = 63 \cdot cS$ . The other lakes can be similarly analysed, to give a set of 5 equations:

$$\begin{aligned}1000 &= 63 \cdot cS \\4500 &= 47 \cdot cM \\1000 + 63 \cdot cS + 47 \cdot cM &= 157 \cdot cH \\3500 + 157 \cdot cH &= 173 \cdot cE \\3000 + 173 \cdot cE &= 208 \cdot cO\end{aligned}$$

It is not hard to work your way down these, solving first ( $cS = 1000/63$  is not so hard to solve ...) and then the 2nd, which then allows you to solve the 3rd, before then solving the 4th and 5th in turn .... However, the system of equations you might have to solve could be (and usually is) much more complicated. Fortunately, we can get **MATLAB** to do the work. :) It may be far from obvious what **MATLAB** has to do with this, so I'll do a little re-arranging of the 5 equations:

$$\begin{aligned}63 \cdot cS + 0 \cdot cM + 0 \cdot cH + 0 \cdot cE + 0 \cdot cO &= 1000 \\0 \cdot cS + 47 \cdot cM + 0 \cdot cH + 0 \cdot cE + 0 \cdot cO &= 4500 \\-63 \cdot cS + -47 \cdot cM + 157 \cdot cH + 0 \cdot cE + 0 \cdot cO &= 1000 \\0 \cdot cS + 0 \cdot cM - 157 \cdot cH + 173 \cdot cE + 0 \cdot cO &= 3500 \\0 \cdot cS + 0 \cdot cM + 0 \cdot cH - 173 \cdot cE + 208 \cdot cO &= 3000\end{aligned}$$

This is starting to look scarily like some matrix stuff. Satisfy yourselves that these two sets of equations are the same, and that all I have done is to write them with the unknowns on the left hand side ( $cS$ ,  $cM$ ,  $cH$ ,  $cE$ , and  $cO$ ) and the knowns (the metal input fluxes) on the right hand side. In fact, this can be written in matrix form:

$$\begin{pmatrix} 63 & 0 & 0 & 0 & 0 \\ 0 & 47 & 0 & 0 & 0 \\ -63 & -47 & 157 & 0 & 0 \\ 0 & 0 & -157 & 173 & 0 \\ 0 & 0 & 0 & -173 & 208 \end{pmatrix} \times \begin{pmatrix} cS \\ cM \\ cH \\ cE \\ cO \end{pmatrix} = \begin{pmatrix} 1000 \\ 4500 \\ 1000 \\ 3500 \\ 3000 \end{pmatrix}$$

Brush up on your matrix maths and check that Eq. 5 is exactly the same as before. It is just the series of 5 separate equations, but represented in matrix math form. Write out the matrix multiplication in full to get the 5 separate equations back again if you are not convinced that this is the case.

In a new **MATLAB** m-file, create a  $5 \times 5$  array containing the values in the matrix on the left hand side of the equation above and assign it to the variable **R** (for River flow). Create a  $5 \times 1$  array containing the vector values on the right hand side of the equation and assign it to the variable **F** (for heavy metal Flux). The solution to this problem is the set of (steady-state) concentrations of heavy metals in the 5 lakes. (Call this variable **C**.) We thus have the equation:

$$R \times C = F$$

If we could determine the inverse of  $R$ , we could write:

$$R^{-1} \times R \times C = R^{-1} \times F$$

(I have simply multiplied both sides of the equation by  $R^{-1}$ .)

Recognizing that a matrix ( $R$ ) multiplied by its inverse ( $R^{-1}$ ) is the Identity matrix ( $I$ ), and that  $I$  leaves everything it multiplies alone, we have:

$$\begin{aligned} I \times C &= R^{-1} \times F \\ \Rightarrow C &= R^{-1} \times F \end{aligned}$$

We are there! We have  $R$  and  $F$ , so by multiplying  $F$  by the inverse of  $R$ , we get our set of 5 solutions (in the  $5 \times 1$  vector  $C$ ). And **MATLAB** will give you the inverse of  $R$  (if it exists) on a plate.<sup>77</sup> Sweet deal!

Now you have everything you need – go solve the steady-state problem for the unknown metal concentrations in the 5 lakes (the vector array  $C$ ) using the inverse of  $R$ . You can always plug these values into the original equations to satisfy yourselves that it all works out.<sup>78</sup>

<sup>77</sup> At the command line; type:

```
» help inv
```

to find out how to get your paws on the inverse of  $R$ . You can also lookup 'inverse of a matrix' in the Index of **MATLAB** Help.

<sup>78</sup> Note that the equations above are written in normal maths language, e.g. with a  $\times$  rather than the  $*$  that **MATLAB** understands.





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. <sup>1</sup> ... If only there was some way of storing all these commands in such a way that they could be worked on and run again with the press of a button (as a wild guess, how about F5?), without having to enter them all in, all over again from scratch ...

Your wish is granted! In **MATLAB**, it is possible to store all of your commands in a single text file, and then request that they (the list of commands) are all executed (sequentially) at one go. **MATLAB** gives this text file a fancy name (because it is a very fancy piece of software, after all) – a *script*<sup>2</sup>, otherwise known as an *m-file*. To create a new *m-file*; from the File menu, select Script (a common type of *m-file*)<sup>3</sup>. 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 save it with the extension `.m`, rather than e.g. `.txt` – this creates you a (script) *m-file* (illustrating that an *m-file* is nothing more than a text file with a `.m` file extension). 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<sup>4</sup>. 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` (remembering that spaces are NOT allowed in filenames). You are going to use the function `disp` (see margin help box and/or type `» help disp` to find out the **MATLAB** *function* syntax and usage). This command (*/function*) will print to the screen, either any text you specify (in inverted commas), or the contents of any *variable* (you pass the variable name to `disp`). For now, simply pass the text directly.

Your program needs just a single line in the *m-file*:

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

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

### *m-file*

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

Assume a similar convention to that for *variables* in the naming of *m-files*.

<sup>2</sup> The conception of a *function*, will be introduced later.

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

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

### `disp`

... displays something (the contents of a variable) to the screen.

In the example of:

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

where `STRING` is a .... *string*, you get the *string* displayed as text at the command line.

You can also pass the name of any variable

```
disp(VARIABLE)
```

and get the contents of `VARIABLE` displayed.

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

```
disp(X)
```

is ... well, find out for yourself!

Note that in some situations, its effect is simply the same as leaving off the semi-colon (`;`) from the end of a line.

Save the file (to your working directory). Run it at the command line by typing its name (omitting the .m extension). Your first program is a success! (Surely you could not screw up a single line program ... ?<sup>5</sup>)

You could extend this to a mighty 2-line program by defining the string as a variable and displaying the contents of the variable, i.e.,

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

(Try this out.)<sup>6</sup>

For further practice – pick one of any of the previous exercises in which multiple lines of code were required, such as loading and then plotting a data set, place these lines 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 by typing its name at the command line (omitting the .m extension).

### 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 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 considered a comment<sup>7</sup>, and is ignored by MATLAB.
- Structure the code nicely. You can break the code up into sections, e.g. by adding a blank line. You might also start each section with a label summarizing that it is going to do (via the addition of a *comment* line).
- To start with – program in as simple a 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 in between.
- Always save your changes before running your program (or you may unknowingly be running the previous version).

<sup>5</sup> If MATLAB gives you an error message something like

```
Undefined function or
variable 'hello_world'
```

then it is likely you are simply not in the same directory as the *m-file*, and/or the location of the *m-file* is not in one of the directory paths MATLAB knows about (see previous Tutorials for comments on changing directory vs. adding paths.).

<sup>6</sup> Remember that when a *function* requires a *string* input, you can either pass the string directly (encased in inverted commas), or assigned the string to a variable, and pass the name of the variable (no inverted commas).

#### Creating help text in an *m-file*

MATLAB allows you to create a 'help' section in the *m-file* – text that is outputted to 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 one 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.

<sup>7</sup> Your % comment can start on a new line, or follow on from the end of a line of code, whichever is more helpful.

- 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. so that if you repeatedly run the script such as you might in developing and debuggin it, you don't end up with 100000000s of Figure windows open ...

An illustration (and a far from perfect illustration) of a short *function* (*m-file*) exhibiting at least a few examples of good practice, is:

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

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

This example also illustrates one possibility for a consistent *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 the use of the semi-colon at the end of every line to prevent (here unwanted) printing of results to the screen. (Don't worry about what a *function* is yet ... just not the degree of commenting and that there is some sort of consistent and meaningful naming convention.)

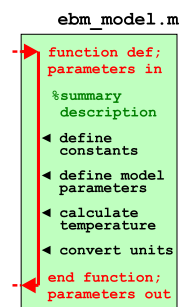


Figure 2.1: Schematic of the example program.

In the file, you can create as much 'ASCII art' as you like if it helps to make the code clearer, e.g. adding separator comment lines ...

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

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

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

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

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

```
...
```

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

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

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

The book schematic structure of this program (*script*) is shown in Figure 2.2.<sup>8</sup>

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<sup>9</sup>.

### 2.1.2 Debugging the bugs in buggy code

What programming is mostly about is not writing new code so much as debugging<sup>10</sup> 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)<sup>11</sup>.

Here is 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`<sup>12</sup>. Then add the following lines to the file:

<sup>8</sup> Note that not all of the comment lines are shown in the structure schematic – only the main program summary at the top.

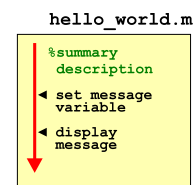


Figure 2.2: Schematic of the Hello World program.

<sup>9</sup> Because it will not work 99 times out of 100 ...

<sup>10</sup> The art of fault-finding in computer code.

<sup>11</sup> And by the discipline of software engineering, which is way out of scope of this course.

<sup>12</sup> Remember – you are advised to name your *m-files* as something vaguely descriptive of what the script actually does (and you do not 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!)

```
66 str = 'do you like bananas?'
```

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

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

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

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

Actually ... if this were your program, you should have paid attention to earlier and not have written it all at once before testing it! But at least **MATLAB** is giving you some sort of feedback. The actual error reported might not always mean that much to you but the line number at which the problem occurred is gold-dust. The line of code it does not like is line 6<sup>13</sup>, which is:

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

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

```
> cos[0.0]
```

to which you get told:

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

Now you have reduced the use of the `cos` command to its simplest, whilst retaining the usage in your program that seemed to cause an issue. Hopefully, now the error is apparent. If still not, check out help

<sup>13</sup> 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.

on the `cos` function, or search `cos` in the **MATLAB** help (from the question mark icon in the toolbar).

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

Other strategies for helping debug include:

1. Checking the what the values of the variables were at the point at which the program derp-ed – the current (and the point of program crash) variable values are listed in the Workspace window.
2. Changing the relevant variable value(s) (here `x`) and re-typing the problem line to see if it makes a difference<sup>14</sup>.
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 of code, you can go comment out the new code (never normally just delete it all), check that the original section of code still works, and then line-by-line, un-comment the new code until the problem line is found.
4. You can also put your program on hold just before the problem line and explore the state of the variables at that point (see Box), although in this particular example of a bug, **MATLAB** does not allow this, presumably because it 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 apparently have a problem on line 12:

```
Undefined function or variable 'k'.
```

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

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

Once you have fixed the second bug; saved, and re-run the script, you should see Figure 2.3. (unless there were further bugs to find ...)

<sup>14</sup> This is sort of similar to the example given of simply testing a specific value directly.

#### Debugging – breakpoints

*Breakpoints* are indicators in the code that tell **MATLAB** to pause at 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 at 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.

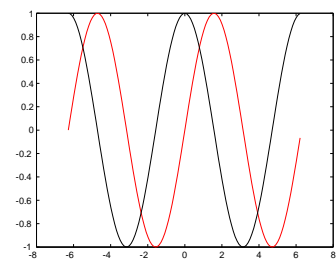


Figure 2.3: Output from the (bug-fixed version of) `plot_some_dull_stuff` *m-file*.

```
68 str = 'do you like bananas?'
```

## 2.2 Functions

*Functions* in **MATLAB**, are really just fancy *scripts*. Again – just plain old lines of code in a text file that is given a .m extension (making it an *m-file*). The big difference from a *script* in MATLAB is that a *function* can take variables as input and/or return variables (or variable values) as an output. (In contrast, a *script* takes no input and returns no outputs, other than plots or data files that might be saved.)

A *function* is defined (and differentiated from a *script*) by a special line at the very start<sup>15</sup> of the *m-file* (see Box). You must follow the **MATLAB** syntax exactly in defining a function.

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<sup>16</sup>. So how about if in your work, you found you frequently needed to use the square of the sine of a number. You could keep writing:

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

or, if you were a little more devious, you could create your own function for returning the square of the sine of a number.

In this example, the contents of your *m-file*, which here we'll call `sin2`<sup>17</sup>, would look like:

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

but of course with lots of comments to remind you what the *function* does etc.

Your new *function* is used pretty much as you would expect and have used previously, e.g.

```
» sin2(0.5)
```

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

```
» Y = sin2(0.5);
```

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

<sup>15</sup> 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 want 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
```

<sup>16</sup> Actually you can, but it is best not to.

<sup>17</sup> And hence filename `sin2.m`.



Go make up your own *function* now. Start by creating one that takes a single input and returns a value equal to the sine of the square of the value (rather than the square of the sine as above). Test it (i.e. compare the output of your *function* with the equivalent calculation typed in at the command line).

When you are happy with this, create one with 2 inputs (refer to **MATLAB** help on `function` and/or refer to the previous Box), that returns a value equal to the sine of the first input, divided by the cosine of the second input<sup>18</sup>, 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 assign the returned values to a variable, you may not have noticed. For example, `plot` and `scatter` are in fact *functions*, and return an ID of the plot graphic. We simply have not been asking for the returned value so far. As per **MATLAB** help:

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

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

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

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

Here, we have created a variable `tmp` to hold the value of the partial calculation. It does not appear in the Workspace window when you use the function. The advantage of this is that you could create a second function that also created a temporary variable internally called `tmp` with both instances of `tmp` treated entirely separate and isolated by **MATLAB** (i.e. setting the value of one instance of `tmp` does not affect the value of the other).

The private nature of *variables* created within *functions* does however lead to some additional complications in debugging *functions* because when the function terminates, you have no record of what occurred during its execution (in terms of not being able to access the value of any of the variables used within the *function*). Try

<sup>18</sup> 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*.

```
70 str = 'do you like bananas?'
```

setting a breakpoint at the start of the line where the square of `tmp` is calculated – note that `tmp` now appears in the Workspace window. Continue the *function* and when it terminates, note that `tmp` is now gone from the list.

## 2.3 Conditionals '101'

### 2.3.1 *if* ...

One of the most important programming constructs is the *conditional statement*, in which whether one or more *statement(s)* are executed (and hence the overall outcome) is conditional on the 'truth' or otherwise (i.e. it being true or false) of a given *expression*.<sup>19</sup>

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

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

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

```
if happy
...

```

where `happy` is a variable.

2. A **MATLAB** *function* returning a true or false, e.g.

```
if isnan(A)
...

```

where variable `A`, may or may not be a NaN.

3. A *relational operator* (see earlier), i.e. one of e.g.:

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

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

```
if A > B
...

```

where `A` and `B` are numbers.

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

---

AN INITIAL AND RATHER COMPUTER PROGRAMMING TEXTBOOK-LIKE EXAMPLE is as follows:

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

But before we worry about anything else (e.g. how to apply a *conditional* statement), you'll need to know about inputting information into a **MATLAB** program from the keyboard<sup>20</sup>. 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).

<sup>19</sup> Pause ... and deep breath.

#### Conditional Statements

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

The basic `if` structure is:

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

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

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

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

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

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

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

<sup>20</sup> All programming languages have such a facility and man basic programs, at least in the Old Days prior to widespread *GUIs*, make use of keyboard input

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

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

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

Run the program thus far. You should see the question displayed, and when you type in an answer and hit **RETURN**, the program will end. Because your *m-file* is configured as a *script* and not a *function* (see earlier), you can see the variable `MY_ANSWER` in the variable list and you can hence check its value – it should contain a *string* with the answer you gave to the question. Make sure it all works like this so far.<sup>21</sup>

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

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

#### input

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

For inputting a numerical value:

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

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

For inputting a string:

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

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

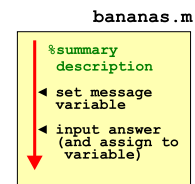


Figure 2.4: Schematic structure of the simple bananas question program.

<sup>21</sup> 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? ';`

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!'). In this (now illegal!) usage, we are trying to ask whether the contents of the *variable* `my_answer`, are equivalent (the `==`) to the string 'yes'.

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

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

Or, we could have made this more compact:

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

Your code should now comprise something like the 3 lines from before (comment, define question, get input) followed by 4 lines of code of the conditional structure, comprising: the `strcmp` function, the `if ...`, use of `disp` to display a message, and lastly, `end`. The structure should look like Figure 2.5<sup>22</sup> or if you assign the message to a 2nd variable, like Figure 2.6. A complete example program ... to help you follow all the above, would look like<sup>23</sup>:

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

#### `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'.)

<sup>22</sup> The red triangle denotes a branch point, where the code can go in different directions depending on the result of the *conditional*. In this example – there is only one branch, corresponding to the answer being 'yes'.

<sup>23</sup> Note the indentation of the contents of the `if ... end` structure. This is very common programming practice. You can make **MATLAB** do this for you by selecting a single line, or highlighting a block of lines, and clicking on the Indent icon in the code editor.

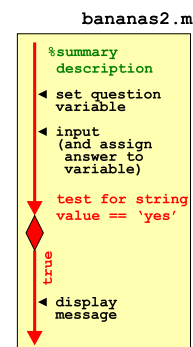


Figure 2.5: Schematic structure of the extended bananas question program.

```

% test the answer ... and reply if 'yes'
if strcmp(var_answer, 'yes')
    disp(var_response);
end

```

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

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

You can also turn this around, and test for an answer that is not 'no' (the `~` is making the test, not 'no'), i.e.

```

if ~strcmp(MY_ANSWER, 'no')
    [MESSAGE]
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'.

Next, you might display an alternative message if the answer is not 'yes'. Refer to **help** / the margin Box on `if ...` and note that you can extend the structure with an `else` which would be followed by a line displaying the alternative message (e.g. 'Then you need to get a life, apple-lover.')<sup>24</sup>.

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

```

if strcmp(MY_ANSWER, 'yes')
    [MESSAGE1]
else
    [MESSAGE2]
end

```

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

```

if strcmp(MY_ANSWER, 'yes')
    [MESSAGE1]
elseif strcmp(MY_ANSWER, 'no')

```

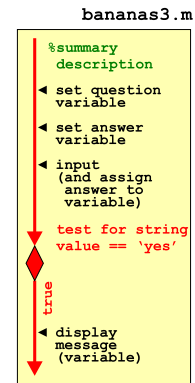


Figure 2.6: A slight variant on the schematic structure of the extended bananas question program.

<sup>24</sup> And then the line with `end` after that – follow the prescribed structure \*exactly\*.

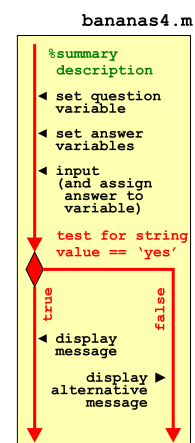


Figure 2.7: Schematic of the bananas program using the `if ... else ...` construct (and displaying alternative messages).

```

    [MESSAGE2]
else
    [MESSAGE3]
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.

---

CONTINUING TO BEAT THIS SAME TIRED EXAMPLE TO DEATH ... what if some wise-crack answered ‘YES’ rather than ‘yes’?<sup>25</sup> One could write:

```

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

```

This will work, but you might note that you have had to exactly duplicate the MESSAGE line. If instead of displaying a simple message, a complex calculation was carried out – all the lines of the code following the `if ...` would have to be exactly duplicated after the `elseif ...`. While it might seem trivial to simply copy-paste the required lines, this is<sup>26</sup> 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.<sup>27</sup> (You could extend it to ‘no’

<sup>25</sup> 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 ...

<sup>26</sup> Note quite in the same way that driving down a mountain highway with your eyes shut or hungry sharks are dangerous.

<sup>27</sup> 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’.

```
76 str = 'do you like bananas?'
```

also but I think you get the point ...) Be careful with all the nested parentheses – an source of mistakes/bugs. You might write it like this, for example:

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

---

A NON-TEXT AND NON FRUIT RELATED EXAMPLE. ALMOST.

How many bananas could you eat in a day? I bet it is less than ten. We'll let the computer ask and if the answer is 10 or more, you (the computer) replies: 'liar!'.<sup>28</sup>

The basic code is very similar to before. Create a new *m-file*, add a comment line, define your question ('How many bananas do you think you could you eat in a single day?') and then get **MATLAB** to ask it and pass back whatever is entered in at the command line. The only difference at this point – refer to the usage of `input` (see earlier Box) – is that we want a number input rather than a string. You can call the *variable* into which you assign the result of `input`<sup>29</sup>, the same as before, or to make it distinct, e.g. `N_BANANAS`, i.e.

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

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

```
if (N_BANANAS >= 10)
    [MESSAGE1]
else
    [MESSAGE2]
end
```

or equivalently:

```
if (N_BANANAS > 9)
    [MESSAGE1]
else
    [MESSAGE2]
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 slightly 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.

<sup>28</sup> 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`  
.....

<sup>29</sup> Note that the usage of `input`, now does not include the parameter '`s`' being passed (which previously specified that a *string* input is expected).

#### 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
```



For instance, and back to the ... fruit ... Consider the situation in which you want the same answer for multiple different kinds of fruit. Trying to code 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 that 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 might instead 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 variable containing 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 values, **MATLAB** requires the list after `case` to be encased in curly brackets: `{}`. For a single answer, it would just be:

```
case 'banana'
```

for a string, and for a number:

```
case 10
```

---

Try re-formulating one of your earlier yes/no `if` questions in the form of a `switch ... case ...` code structure, with multiple possibilities for the form of e.g. a 'yes' dealt with by:

```
case {'YES', 'yes', 'Y', 'y'}
```

## 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 or more lines of code<sup>30</sup>), 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 through an example, almost as contrived as for conditionals :o)

### 2.4.1 For ...

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

---

**LOOPS GROUND ZERO.** Basically – `for` loops cycle through a series of numbers between specific limits, or if you like, 'count' up (or down) through a series of numbers. As the loop counts (cycles), it allows you to execute some code, so for each count (or cycle), the (same) block of code is executed. We'll worry about what you might 'do'<sup>31</sup> (i.e. the code fragment) in a *loop*, later.

Consider, or rather: create a new `m-file`<sup>32</sup>, and add following code for a simple loop to the file:

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

Save it. Run it. What did it do?

I bet you have absolutely no idea! It actually cycled around ten times, counting from  $n=1$  through  $n=10$ , but you would not know it as there was no code within the loop to do anything and tell you anything about it.<sup>33</sup>

There are 2 alternative but very crude debugging strategies you could take<sup>34</sup>:

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

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

#### loops in MATLAB

##### for

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

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

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

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

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

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

##### while

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

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

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

<sup>30</sup> 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.

<sup>31</sup> Note intentionally a joke. Actually, this is only funny if you know **FORTRAN**, and even then it is only marginally funny.

<sup>32</sup> Comment it!

<sup>33</sup> 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.

<sup>34</sup> Plus, you could add a *breakpoint* and view the value of `n` in the *Workspace* window each cycle around the loop.

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

2. Or you can print the value of  $n$  'properly'<sup>35</sup>, e.g.

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

You could tart this up further by creating a string that provides more explicit information back to you, which is when you really need to use `num2str`, e.g.

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

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

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

(but they work the same – check it).

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

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

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

---

LOOPS IN ACTION. So, consider the following (somewhat 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).<sup>36</sup> The basic code is simple and you can try it out by first creating a new (*script*) m-file.

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

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

The first 4 lines you should be A-OK with, you have seen something very like this before. In the last line, again, 2 strings have been

<sup>35</sup> Although you can get away with just writing:

```
disp(n)
```

<sup>36</sup> Obviously, one way to do this would be to enter the numbers into a file first, use the load function, and calculate the sum.

```
80 str = 'do you like bananas?'
```

*concatenated* by enclosing 'The sum of the numbers is: ' and `num2str(A+B)` in a pair of brackets `[]`. The string representing the number sum is itself created by adding A and B, and then converting the resulting number into a string using `num2str` (see earlier). As always – if you are happier breaking down the last line into its component parts, e.g.

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

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

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

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

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

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

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

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

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

(noting that in your own code, `MAX_N`, for instance, could be 4).

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

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

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

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

Two things to be aware of in doing this:

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

```
y = [];
```

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

---

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

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

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 and 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 starts and that you then try and add `input(prompt)` to. The solution is to initialise the value of *x* before the loop starts, e.g.

```
x = 0;
```

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

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

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

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

```
82 str = 'do you like bananas?'
```

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

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

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

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

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

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

#### 2.4.2 Other loop configurations and usages

In the previous examples, the *loop* limits were fixed in the program itself – you’d have to edit the *script* code and re-save the file in order to be able to input and sum a different number of values. You could create a more flexible program by making the m-file a *function* rather than a *script*.<sup>37</sup>

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

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

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

```
» function_sum(5)
```

will request 5 inputs in turn, and at the end, display the sum.<sup>38</sup>

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

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

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

Why? (Try it and see, even.)

<sup>37</sup> There are other ways of adding flexibility to the loop count that we’ll see shortly.

<sup>38</sup> So in addition to the code fragment given, you need to define (at the top) and then end (at the bottom) a *function*, you need to create a running sum, and then after the *loop* finishes, display the sum.

Alternatively, you could write your program as a *script* and before the loop starts, 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.

---

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

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

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

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

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

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

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

### 2.4.3 *Fun(!) worked examples*

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

---

*Loops, CAMERA, ACTION!*<sup>39,40</sup> (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)<sup>41</sup>. 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.

<sup>39</sup> **Example codes provided**

<sup>40</sup> (at end of text)

<sup>41</sup> In scripting, it is also possible to automate downloading files from the internet.

```
84 str = 'do you like bananas?'
```

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(?)

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`<sup>42</sup>. We should start by loading this little lot in. For the first file we could write:

```
» temp = load('temp1.tsv','-ascii');
```

or equally:

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

Or if we wanted to save all the data in a 3D array, we could also write:

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

Can you see that these statements are identical? Try them all out, just to be sure. The last form is really useful, because we can now go on and write:

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

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

You could go on and load in the March, April, etc data in a similar fashion, but you should be able to see a pattern forming here – each filename differs only in the number at the end of its name and this number corresponds not only to the number of the month, but will also correspond to the layer index of the 3D array that you will create. This is something that a *loop* could be used for while you go off for a coffee. So this is what we are going to do – use a *loop* to load in all of the files. So go back and delete the lines that load in the files, one-by-one.

**Create a new script m-file.** Call it ... anything you like<sup>43</sup>. 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.

<sup>42</sup> Don't worry about the `.tsv` file extension – the file format is plain old text (ASCII) and could have instead been `.txt`.

<sup>43</sup> `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.



We first need to construct the *loop* framework. We'll call the month number counter variable, `month`. Create a *for loop* (with nothing in it yet) with `month` going from 1 to 12.<sup>44</sup> 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<sup>45</sup>. 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<sup>46</sup>). Within the loop, add the following line:

```
disp(month)
```

and then save and re-run the *script*. Now you can see how the loop progresses. This sort of thing can be useful in helping to *debug* a program – it allows you to follow a program's progress, and if the program (or **MATLAB** script) crashes, then at least you will know at what loop count this happened at, even if you are not given any more useful information by **MATLAB**. Only when you are happy that you have constructed a *loop* that goes around and around 12 times with the variable `month` counting up from 1 to 12; comment out (%) the printing (`disp`) line<sup>47</sup> (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* separate strings. For example:

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

will create the filename out of 3 components parts – a common elements of all the filenames ('temp'), the number of the month ('1'), and the file extension ('.tsv').

2. Converting a number value of a (count) variable to a string (the `num2str` function), so instead of hard-coding in the string representing a number (1 in this example), you convert from the value of a counter, e.g. `num2str(month)`.

This is where the role of the loop counter (stored in the variable `month`) comes in. Each time around the loop, the value of variable `month` is the number of the month. All you have to do is to convert this value to a *string* and thereby automatically generate the correct month's filename each time (as per above).

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

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

<sup>44</sup> 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 %.

<sup>45</sup> Typing: the `m-file` filename without the extension.

<sup>46</sup> 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.

<sup>47</sup> 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.

#### `num2str`

Converts a number to a *string* (`s`), e.g.

```
s = num2str(N)
```

where `N` is any number type variable.

`num2str` is useful in adding specific captions to plots (with caption text based on the value of a numerical variable) and in creating automated strings (e.g. filenames) within a loop.

86 `str = 'do you like bananas?'`

and after it (still within the *loop*) some debugging<sup>48</sup>:

```
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.

Now *comment out* (or delete) the `disp(filename)` line, and add a new line to load in each dataset from the filename that is constructed each time the loop goes around.<sup>49</sup> e.g.<sup>50</sup>:

```
temp(:, :) = load(filename, '-ascii');
```

Note that rather than specifying the filename explicitly in the load command, you are now passing the string contained in the variable `filename`. (Hopefully on the previous line of code within the loop, you have created the string value of `filename` ...)

---

We'll now add some graphics.

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

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

(or could be simply written `pcolor(temp);`) and should produce extremely exciting graphics as in Figure 2.8<sup>51</sup>.

Save and run the *script*. Do you have 12 different temperature plots on the computer screen?<sup>52</sup> Note that if you keep running the program, you'll get 12 more figure windows each time. This is where the `close all` command comes in useful, and you could add this at the start (or end) of your *script*. Because if you re-run the *script*, you wont then end up with 24 figure windows. And then 36 the time after that, and ...

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

---

Finally ... look up **MATLAB** help on `getframe`. Then go back to your global temperature loading/plotting script and add the following line to your program<sup>53</sup>:

<sup>48</sup> Or you can make use of a **breakpoint**.

<sup>49</sup> 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.)

<sup>50</sup> Alternatively, you could store all the data slices as you load them in and rather than specifying the 1st, then 2nd, etc layer of the 3D array, here we are specifying the layer with an index equal to the contents (or value) of `month`, which, if you remember, counts up from 1 to 12 in the *loop*.

```
temp(:, :, month) = ...  
load(filename, '-ascii');
```

If you run this coding variant and take a look at the Workspace window – note that you have an array (`temp`) that has size  $94 \times 192 \times 12$ .

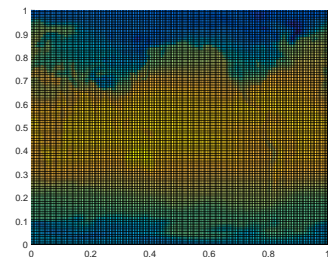


Figure 2.8: Extremely unappealing blocky plot of Earth surface temperature (who cares with month? – the graphics are too poor to matter ...).

<sup>51</sup> The 2D graphics will get \*much\* better later – one thing at a time!

<sup>52</sup> If not, stick you paw up in the air for help ...

<sup>53</sup> Where to put the line? Within the loop and after you have plotted the plot.

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

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

```
» movie(M,5,2)
```

and hopefully ... an animation of the progression of monthly surface air temperatures globally, should appear<sup>54</sup>.

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 the course? What about next year? Are the *loop* contents indented? It is important that it is commented and laid out adequately.

---

Creating a portable animation format would be useful (i.e. that you could play on a different computer or upload to the www). There is no longer a convenient **MATLAB** command to turn the **MATLAB** format movie (M) into a format you can use elsewhere (there used to be a command called `movie2avi` (see Box), but it has been ... retired (curse you, Mathworks)). The new/replacement command is `VideoWriter`, which differs mostly in that the animation is now created within the program and the .avi format animation has its frames added (within the loop) as the graphics are created (rather than converting the frames afterwards as in `movie2avi`).

To use the **MATLAB** `VideoWriter` function, you will need to adjust your code along the following lines (and as per the **MATLAB** help on `VideoWriter`)<sup>55</sup>:

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

<sup>54</sup> Note that the active Figure window may have disappeared behind some other windows so go rescue it to see what is happening.

<sup>55</sup> Do not overly worry about the exact usage/syntax of `VideoWriter`.

```
88 str = 'do you like bananas?'
```

```
% Close the file.  
close(vidObj);
```

Note that the code above does not bother to create a 3D array of all the data loaded in, but rather, over-writes a 2D array (`temp`) each time around the loop (having first created the animation frame). An animation is created at the end, but there is no record of all the 12 monthly datasets that were loaded in (and this may well be correct/the point). (See the previous margin comment on how to modify things to create a 3D array of all the data.)

#### **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.

## 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. But this can all also become confusing ... remember to indent your code.

### 2.5.1 for ... and conditionals

Firstly, one might (rather trivially) use a conditional to decide whether to execute a loop 10 or 100 times, e.g.

```
my_string = input('Loop only 10 times (y or n)', 's');
if strcmp(my_string, 'y')
    for n = 1:10
        SOME CODE
    end
else
    for n = 1:100
        SOME CODE
    end
end
```

Here, we have a *conditional* structure testing whether the string entered in response to the questions is 'y'. If 'y', then a loop of maximum count 10 executes, if not ('y') (else), then a loop of maximum count 100 executes.

This is a little messy and could be cleaned up and simplified somewhat. For instance – by replacing the maximum loop count as a variable, whose value is set by the (*conditional*) answer. e.g.

```
my_string = input('Loop only 10 times (y or n)', 's');
if strcmp(my_string, 'y')
    n_max = 10;
else
    n_max = 100;
end
for n = 1:n_max
    %
end
```

Enter in the second code (or some variant of it) into a new *script* m-file, and explore how it works – try changing the alternative loop limits, add a line within the loop to `disp` the value of `n` and hence confirm that the correct number of iterations of the loop occurs.

Note that you will need to replace `%` with your own line (e.g. using `disp`) or you could have nothing and leave the comment `%` line in. Remember to add comment lines.

---

#### 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 indentation (or lack of) does not affect the execution of the code (unlike in e.g. **Python**).

```
90 str = 'do you like bananas?'
```

Returning to the previous loop example concerning summing a series of numbers entered – 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:

```
% set up some strings for the 2 questions
my_question1 = 'Please enter a number: ';
my_question2 = 'Another input (y/n)? ';
% initialize running sum
running_sum = 0.0;
% START OF LOOP
for n = 1:1000000
    % ask for a number
    my_number = input(my_question1);
    % update running sum
    running_sum = running_sum + my_number;
    % display running sum
    disp(['sum so far = ' running_sum])
    % ask whether to keep going
    my_string = input(my_question2,'s');
    % exit loop if answer is no
    if strcmp(my_string,'n')
        break
    end
end
% END OF LOOP
```

where 1000000 in the code is simply chosen as a 'very large number' and one rather larger than the maximum number of numbers you could ever imagine entering<sup>56</sup>.

The key new command here is `break`. The way the code works (hopefully!) is that towards the end of the *loop*, the 'another input' question is asked – if no further input is required, the loop exits via the `break` command. Remember that now we have *loops* and *conditionals* nested together, it helps even more to *indent* the code<sup>57</sup>. 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 text could equally be placed directly within the *loop* within the call to the *input function*.

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
```

<sup>56</sup> There is a better way of doing this, with the `while` construct, that we'll see shortly.

#### `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.

<sup>57</sup> **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.

```

my_number = input(my_question1);
my_string = input(my_question2, 's');
if ~strcmp(my_string, 'y')
    break
end
end
end

```

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

**Enter in the (highlighted) code into a new *script* m-file and modify as per above, such that `break` is executed if any answer other than `'y'` is entered.**

A PRACTICAL EXAMPLE of testing the value of a *variable* and breaking out of a *loop* depending on the result of the test, would be when saving a data file. You might test for a filename that already exists and if so, automatically modify the new file name so as not to overwrite the existing file.<sup>58</sup> The relevant function is `exist`, and in the case of a test for a file, the function 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, 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<sup>59</sup>. If not, and the filename has not already been used, you can proceed as before – the equivalent of ‘doing nothing’.

You could start by defining a default filename in the code<sup>60</sup> 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 you are going to 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.

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

where here, we take the string contained in `my_filename`, we append a `'NEW_'` to the start<sup>61</sup>, and assign the new (longer) string back into the variable `my_filename`, like this:

<sup>58</sup> 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.

<sup>59</sup> 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.

<sup>60</sup> Either near the very start of the program (neater), or just before you need to use the string (to save a file).

<sup>61</sup> Note that because the filename already has its `.avi` extension attached, you'll have to modify the start of the string.

```
92 str = 'do you like bananas?'
```

```
if filename_check == 2
    my_filename = ['NEW_' my_filename];
else
    % DO NOTHING
end
```

**See if you can modify the .avi video creating code. An example code for the basic (non filename-checking) program is given at the end of the text. Create a new *script* m-file with this code (or your own), test whether it creates an animation successfully in the first place, and then try and modify it as per above with the filename check. Hints: all the new code needs to go before the line:**

```
vidObj = VideoWriter('my_animation.avi');
```

In this line, the variable `my_filename` that you have defined a default name in, will be passed instead of `'my_animation.avi'`<sup>62</sup>, and the testing of whether your default filename already exists and whether to create a new(modified) filename, all comes before this. The order of the required lines of new of code is:

1. Set default filename.
2. Test whether this filename already exists and assign to the variable `filename_check`.
3. Test whether the value of `filename_check` is 2 and if so, modify the filename.

### 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 margin text on *loops*, 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 new *script* m-file and run it**<sup>63</sup>:

```
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 – try it:

<sup>62</sup> Remember, you can pass a string directly, in which case it must be in inverted commas, or you can pass the variable name. Do not place a variable name in inverted commas (or else the variable name itself will be interpreted as a string, when it is the contents of the variable you want).

<sup>63</sup> You ... are going to need a Ctrl-C on this one ...



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

More interesting and useful is when the statement might change in value as the loop progresses.

Think about the following code (and **type up in a new script m-file and run it**):

```
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  $n = 1:10$ ) (**try this**).

You could also do the counting 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). **Also thy this modification, where the value of  $n$  counts down.**

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 a value of zero (but not below). **Spend a little while playing about with different while configurations and loop criteria, adding `disp` lines or *breakpoints* to find out how many times the loop executes in total.**

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`, the program could be re-coded using a *while loop*:

94 `str = 'do you like bananas?'`

```
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. Here, the 2 lines of input code will keep being executed, as long as the value of `my_string` is `'y'`. Note that in this example, we need to initialize the value of `my_string` (to `'y'` – assuming that we want at least one number). **Try modifying (along the lines of the above) your previous code which was based on a `for` loop, now using `while`.**

---

FINALLY ... WE COULD UPDATE THE FILENAME CHECKING EXAMPLE ... USING `WHILE`. 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 to, rather than append a pre-defined string (`'_NEW'`) to the filename, request that the user provide a completely new filename.

A complete solution would be to address the situation when asking for an alternative filename ... if that file existed too. We could keep checking for a filename clash and keep asking for a new filename, until a unique (unused) filename was provided by the user. Who knows how many attempts this might take (to find an unused filename), so `while ...` would be a better choice of loop than `for ...`. Because `exist` returns a 2 if the file already exists, a logical condition for `while`, would be that a filename determining *loop* continues while `exist` is returning a value of 2. e.g.

```
my_question = 'Enter a filename: ';
while (filename_check == 2)
    my_filename = input(my_question, 's');
    filename_check = exist([my_filename], 'file')
end
```

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

```
filename_check = 2
```

In this case, we do not need a default filename as the user provides a filename on the very first iteration of the loop.

Try it out – add the following code to the start of your basic avi file format saving movie program (e.g. as per at the end of the text), and use the value of the variable `my_filename` as the name for saving the avi animation file and test it.

```
my_question = 'Enter a filename: ';
filename_check == 2;
while (filename_check == 2)
    my_filename = input(my_question,'s');
    filename_check = exist([my_filename], 'file')
end
```

## 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 on top. Currently, and particularly with the very basic 2D plotting you have seen so far (`pcolor`), you are to some extent left guessing where the land and where the ocean is. We are going to work through using a loop to process data that define series of line segments that make up the outlines of the continents.<sup>64</sup>

A pair of files are provided (from the website) that comprise 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 then load these into the **MATLAB** workspace (in the 'usual way'). You should now have 2 vectors. Maybe view them 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 and 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

<sup>64</sup> Example codes provided

96 str = 'do you like bananas?'

sizes of labels can also be increased if necessary. You might end up with something like Figure 2.9.

You can try this at the command line (no need at this point to put everything in a *script m-file*):

```
» lon = load('continental_outline_lon.dat','-ascii');
» lat = load('continental_outline_lat.dat','-ascii');
» scatter(lon,lat);
» axis([-180 +180 -090 +090]);
» xlabel('longitude','fontsize',15);
» ylabel('latitude','fontsize',15);
» title('Continental outline','fontsize',18);
```

(but with lots of comment lines!).

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

Well ... interesting. If you think about it, as one continental outline is completed, the next lon-lat pair will be for the next continent or island. What `plot` does is to join up *\*all\** the adjacent x-y (lon-lat) pairs and hence points, which is why you get the straight lines crisscrossing 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

i.e.:

```
» lstart = load('continental_outline_start.dat','-ascii');
» lend = load('continental_outline_end.dat','-ascii');
```

(Note that you cannot simply call the second variable `end`.)

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 (`lstart`), and ends on row 100 (`lend`). 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:

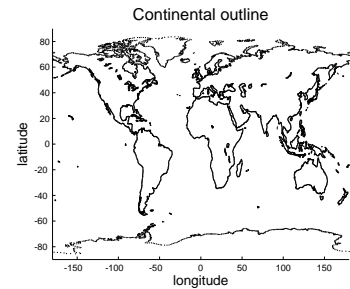


Figure 2.9: Continental outline (of sorts).

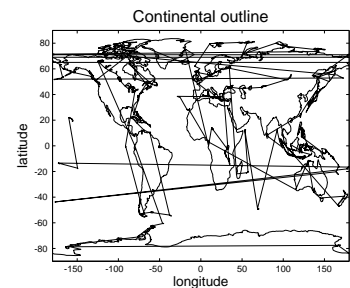


Figure 2.10: Another continental outline (of sorts).

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

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 `lstart` or `lend`.) This is, of course, why we need to get clever with a *loop* and automatically go through all 283 fragments, plotting them on on top of another in the same figure.

How? First you need to write the `plot` command in a more general form – you do not want to have to read the values out of the `lstart` and `lend` 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)<sup>65</sup>. 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<sup>66</sup> and hence loop through all the line fragments. Yes? For instance, setting  $n=1$ , and `plot` (with  $n$  replacing 1 in the code fragment above) – you should again get that very first fragment. Try setting  $n=283$  and `plot`. Do you get the last fragment (what is it of<sup>67</sup>)?

So ... **create yourself a new m-file**. Load in the lon-lat pairs as vectors (renaming them to something more manageable if you wish). Load in the vectors containing 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

#### 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).

<sup>65</sup> You cannot use the obvious variable name `end` – why not?

<sup>66</sup> 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 of 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 ...

<sup>67</sup> An island at about 20N and -150E if you have done it correctly.

```
98 str = 'do you like bananas?'
```

counter). Set axis dimensions and label nicely (after the loop ends). Run it. Hopefully ... something like Figure 2.11 appears(?)

An example code is given at the end of the text should you need any guidance as to e.g. in what order or where certain lines should go.

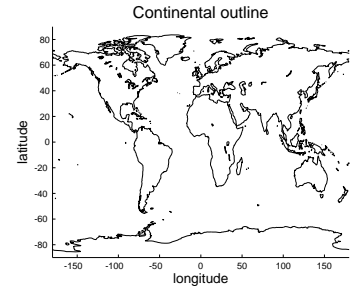


Figure 2.11: Another go at the continental outline!

# 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 the previous chapters.

```
100 str = 'do you like bananas?'
```

### 3.1 Further data input

Previously, you imported ASCII data into **MATLAB** using the `load` command<sup>1</sup>. 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, your 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.

Please regard this section as a simple overview on some of the different **MATLAB** file loading/input options (so that you know that they exist). For many one-off data loading problems, it can be easier to use the **MATLAB** GUI and the Import Data wizard.

#### 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 from 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(!!!)<sup>2</sup>.

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) – this is the value returned by the *function* `fopen`. The *function* you are going to use to actually read the data (having opened the file) is called `textscan`.

3. Once the data has been read it – close the file using `fclose` (see Box). You are going to have to pass the ID of the file again when you call this *function* (so **MATLAB** knows which file to close).

<sup>1</sup> Or maybe 'cheated' and used the **MATLAB** GUI ...

#### 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).

<sup>2</sup> This is very common across all(?) programming languages.



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 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.

---

AS AN INITIAL EXAMPLE to illustrate this alternative (and more flexible) means of importing of ASCII (text) data, we are going to return to the paleo atmospheric CO<sub>2</sub> proxy dataset file – `paleo_CO2_data.txt`. 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’.

OK – so having seen the format of the data in the ASCII file, you are going to work through the following steps (you can do all this at the command line, or add the lines one-by-one to a *script* m-file if you prefer)<sup>3</sup>:

1. First ‘open’ the file – you will be using the *function* `fopen`, and passing it the filename<sup>4</sup> (including the path to the file if necessary, i.e. if the **MATLAB** working directory does not also contain the file). So that you can easily refer to the file that you have opened later, assign the output of `fopen`<sup>5</sup> 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 specifier per element of data.

In this particular Example, there are 4 items of data (per row) – each of them is either an integer or a floating point number<sup>6</sup>, 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');
```

<sup>3</sup> 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**.

<sup>4</sup> 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).

<sup>5</sup> The output is a simple integer index, whose value is specific to the file that you have opened.

<sup>6</sup> At least, none of them are clearly strings, right?

102 str = 'do you like bananas?'

Here, the '%f %f %f %f' bit specifies that the data format consists of 4 floating point (real) numbers.

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

```
» fclose(openfile_id);
```

4. Actually, it does get darker before the light at end of the tunnel ... what `textscan` actually returns – the data that was read in, is placed into an odd structure called 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.

```
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<sub>2</sub> 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<sub>2</sub> proxy dataset. The file is called `paleo_CO2_data.dat` (rather than `.txt`) 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<sup>7</sup>. 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++**<sup>8</sup>. So go open the file with this instead<sup>9</sup>. 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 manually edit out the headers with `comments (%)`<sup>10</sup>, you are still left with the problem of mis-matched columns. You could edit the file in **Excel** to remove the problematic columns as well ... 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 you did before (using `fopen`) and assigned the ID returned by the function to a variable `openfile_id2`.

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 **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 separate arrays.

<sup>7</sup> If you use a **Mac** (or **linux**) however, all text editors should display the content just fine.

<sup>8</sup> Conveniently installed on the Watkins computer lab computers.

<sup>9</sup> Right-mouse-button-click over the file, then select **Open with** and then click on **Notepad++**.

<sup>10</sup> Recall that **MATLAB** ignore lines starting with a % and this includes loading in data lines using `load`.

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:

```
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).<sup>11</sup>:

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

The weird bit translates to ... `%*` == ignore field ... until the line end == `[\n]`, and then read repeat for the next line.

(You are not expected to know or remember this nor be tested on it ... just to park at the back of your mind, that there are flexible ways of dealing with data input, including not reading everything in!)

3. Now you can close the file:

<sup>11</sup> 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]`).

104 str = 'do you like bananas?'

```
fclose(openfile_id);
```

4. ... and convert the results to something 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<sub>2</sub> concentration value (units of ppm). :)

The complete code (**enter into a new script m-file and try it out**) looks like:

```
openfile_id2 = fopen('paleo_CO2_data.dat');
my_data = textscan( ...
    openfile_id2, '%d %d %*[\n]', 'Headerlines', 3);
fclose(openfile_id);
my_data_array = cell2mat(my_data);
```

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 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 ... :)

---

RETURNING TO THE PALEO PROXY CO<sub>2</sub> DATA ... 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<sub>2</sub> data from `paleo_CO2_data.xlsx`.

#### `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**).

Note that the simple usage of the `xlsread` function gives you an *array* containing just the numeric data. If you were to type:

```
» [num,txt,row]=xlsread('paleo_CO2_data.xlsx');
```

then you will get the numeric data returned in the array `num`, but you also get 2 cell arrays<sup>12</sup> – `txt` and `row`. `txt` contains just the text data, and `row`, 'everything'. View these in the Variable window (by double-clicking on the variable names in the **MATLAB** Workspace window).

If you happen to have an **Excel** file with data (of any sort) in it from another class, practice loading in its contents into **MATLAB**. Note that if the **Excel** file contains cells with text in, you'll need to use the more advanced format of `xlsread` (see `Box` or `help`). Also try loading into only a single sheet of an **Excel** file (assuming the file has multiple sheets).

### 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.

**MATLAB** actually has a quick and simple, and ... a long-winded formal way of accessing data in a *netCDF* file:

1. Using `ncread`, which reads data directly from the file.

`ncread` is by far the simplest way, although it lacks in flexibility and deviates from standard practices used across other programming languages.

2. Via a series of function calls to the *netCDF* library.

In the formal and more long-winded approach, you open the file and receive 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 same ID). The *netCDF* standard is also 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. The general approach is as follows:

- (a) Open the *netCDF* file by

<sup>12</sup> If you need to index a cell array, you do so 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 this example.

#### `ncread`

In its simplest incarnation:

```
data = ...
ncread(filename,varname)
```

where `filename` is the name of a *netCDF* file, and `varname` is the name of the data variable in the *netCDF* file.

e.g. if there was a variable called `rain` in the file `climate.nc`,

```
data = ...
ncread('climate.nc','rain')
```

would read the values in the *netCDF* file variable `rain` and assign to the variable `data`.

**MATLAB** provides a couple of further tricks, allowing you to read sections of the full *netCDF* variable data array, or sample the data array – see `help`.

106 str = 'do you like bananas?'

```
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).

(b) This is the weird bit, as we cannot ask for the data we want automatically :o) Instead, given that we know<sup>13</sup> 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<sup>14</sup>.

(c) 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 that **MATLAB** knows which *netCDF* object you are referring to (you are allowed to have multiple *netCDF* files open simultaneously).

---

FOR A *netCDF* EXAMPLE, we'll take the output of a low resolution climate model. 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<sup>15</sup>.

OK – we'll start by opening the file. The ID of the variable we want to extract and plot is called 'grid\_topo'. To load/extract the 2D field and assign it to a variable `data`:

```
>> data2d = ncread('fields_biogem_2d.nc', 'grid_topo');
```

You should now have an array called `data`. It should be  $36 \times 36$  in size. **Plot it**<sup>16</sup>. Can you deduce what it might be of? Is it in the correct orientation? (If not – correct it, by rotating the array, and/or flipping the rows or column.)

<sup>13</sup> There are ways of listing the variables if not.

<sup>14</sup> 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).

<sup>15</sup> 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 ...

<sup>16</sup> Your choice of 2D plotting function that you have already come across, e.g. `pcolor` or `image`, although not all work as well on this particular dataset (e.g. the auto scaling in `image` causes issues).

(There are more appropriate ways of plotting this, which we will encounter later.)<sup>17</sup>

The variable names of other data-sets that you might load and experiment with in terms of plotting function, color scale, and any 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<sup>-1</sup>).
- `sed_CaCO3` – weight % 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 will need to download to experience these wonders, is called `fields_biogem_2d.nc`<sup>18</sup>.

To load the variable 'atm\_temp':

```
» data3d = ncread('fields_biogem_2d.nc','atm_temp');
```

How many dimensions does this array have?<sup>19</sup> What are the lengths along each dimension? Can you deduce which of the dimensions, time might be?

**Plot a lon-lat slice.** Note that you need to select all longitudes and all latitudes in the array, but only one time index.

**Finally – to test your understanding to date, create an animation of how the surface air temperature in the model evolves over time.**<sup>20</sup>

<sup>17</sup> Missing here 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* is 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 would need to use the function `meshgrid`. (We'll re-visit this example once you have seen `meshgrid` in action.)

<sup>18</sup> The back-story is that this contains the 2D surface ocean and atmosphere fields from 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.

<sup>19</sup> e.g. use `size`, or ensure that the Size column in the Workspace window is selected.

<sup>20</sup> You have everything you need – the *vector* of years, and from this you can determine how many different time points (and 2D data slices) 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 regarding 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`.

#### 3.2.1 Contour plotting

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. The graphic produced from this particular website is not particularly great, and you will learn to do at least as good as this.

---

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  $1^\circ$  (longitude and latitude) grid. You could view the resulting elements of the 2D array in the Variable window if you like ... but at  $360 \times 180$  in size, there may not be much of use you can glean by visually inspecting the matrix<sup>21</sup>.

After loading the data file into the **MATLAB** workspace, try throwing the array into the `image` function (which you saw previously) see what happens, e.g.

#### 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).

#### imagesc

For a data array (matrix) `A`,  
`imagesc(A)`  
displays the data array as if a bitmap, but unlike `image` (see earlier), “uses the full range of colors in the colormap”.

<sup>21</sup> 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.



```
» image(etopoldeg);
```

(hopefully something like Figure 3.1). If it had happened to come out displayed upsidedown<sup>22</sup>, then you'd need to first (before plotting using `image`) flip the matrix upsidedown using the command:

```
etopoldeg=flipud(etopoldeg);
```

(and then re-plot using `image`), and if the Earth instead appeared on its side you will need to swap the rows and columns ( $x$  for  $y$  axis):

```
etopoldeg=etopoldeg';
```

using the `transpose` function. 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, **MATLAB** provides a variant of this plotting function, `imagesc` (see Box and/or **MATLAB** help), that calculates the color scale to exactly span the min/max values in the data. Try this alternative plotting function (and get something like Figure 3.2 hopefully).

The function `imagesc` also enables the range of data values the color 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<sup>23</sup>.

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).

At the command line, try out setting a different *colormap*, e.g.

```
» colormap 'pink'
```

and re-then re-plot the global topography data. Try out various different color maps/scales. 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 axes 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

<sup>22</sup> It doesn't in this particular case.

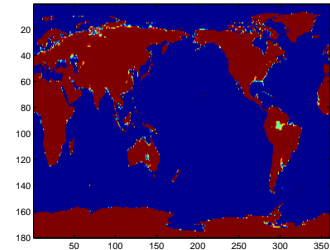


Figure 3.1: Very basic imaging (`image`) of an array (2D) of data – here, global bathymetry.

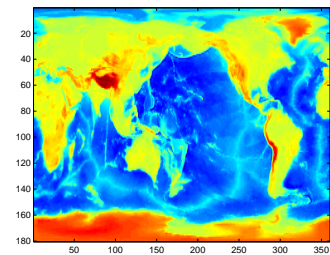


Figure 3.2: Slightly improved very basic imaging (`imagesc`) of bathymetry data.

<sup>23</sup> 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
```

```
110 str = 'do you like bananas?'
```

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, e.g.

```
» contour(etopoldeg);
```

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 (depending on your chosen colormap) and may need adjusting.

Once you have fixed the orientation of the topography map, you might play about with the color scale (`colormap`) as before. You might also try the companion to `contour`, called `contourf`. Also try 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<sup>24</sup>.

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(?)<sup>25</sup> 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.<sup>26</sup>

Having created a specific vector of contours to plot, try it out, by:

```
» contour(etopoldeg,v);
```

(see Box for syntax).

OK – so this is a little weird and maybe not so useful, but you get the point hopefully. Try (still at the command line) 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.

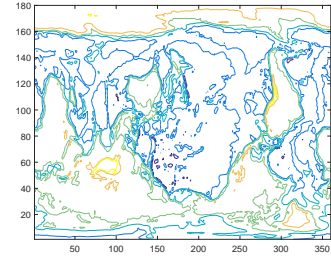


Figure 3.3: Example result of basic usage of the `contour` function.

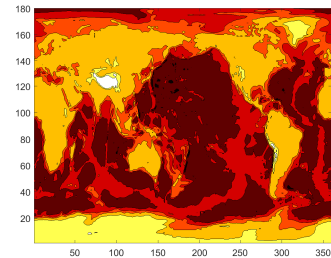


Figure 3.4: Example usage of `contourf`, with the `hot` colormap (giving dark/brown colors as deep ocean, and light/white as high altitude).

<sup>24</sup> By default: **MATLAB** determines the minimum and maximum data values, and draws 10 equally spaced contours between these limits.

<sup>25</sup> If not, it is:

```
» v = [-1000:100:1000];
```

<sup>26</sup> Why, for instance, can you not simply write:

```
» v = [-1000:1000];
```

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

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'? Your 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 slight complication. `clabel` needs to know details of the contours and graphics object with which to do anything with. For the purposes of this course, you don't have to worry about the details of this ... but simply need to note and remember 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 plotting function, using the syntax:

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

which saves a matrix of data to `C`, and a ID (technically: graphics object 'handle') to `h`.

You can test what has been returned by typing:

```
» [C,h] = contour(etopo1deg,v);
```

and looking to see what new variables, if any, have appeared in **MATLAB** Workspace.

2. When you call `clabel`, pass these parameters back in, e.g.

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

(in its most basic usage). So in the example above:

```
» clabel(C,h);
```

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];
```

(which creates 100 m spaced contours between -1000 and +1000 m) maybe you only want labels on contours every 500m, so you'd create a different vector:

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

```
contour(Z)
```

where `Z` is a matrix. This ends up similar to image except with the data contoured rather than plotted as pixels (the 'similarity' 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)
```

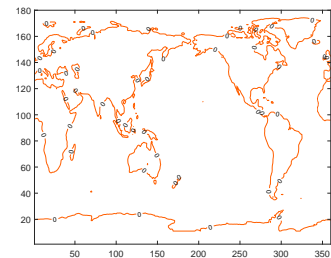


Figure 3.5: Example usage of `contour`, contouring only the zero height isoline, and providing a label.

```
112 str = 'do you like bananas?'
```

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

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

```
» v = [-1000:100:1000];  
» w = [-1000:500:1000];  
» [C,h] = contour(etopoldeg,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 as simple as typing:

```
» 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 ...

#### **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)
```

#### **colorbar**

This almost could not be simpler:

```
» colorbar
```

plots the color scale! By default, it places it to the RH side 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 :))

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. 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<sup>27</sup> ... `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`) (at the Command line):

```
» [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]`.

In our Example – start by noting that the topography data is on a regular 1 degree grid starting at  $0^\circ$  longitude. Latitude starts (at the bottom) at  $-90^\circ$  and goes up to  $+90^\circ$ . We need a matrix containing all the longitude values from  $0^\circ$  to  $359^\circ$  and latitude from  $-90^\circ$  to  $89^\circ$ .<sup>28</sup> These matrices need to be the same size as the data matrix.

<sup>27</sup> **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.

<sup>28</sup> 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).

Maybe just do it and then understand what has happened 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 fully 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 (`etopoldeg`). Re-load the topography data in case you have flipped it about in all sorts of odd ways, and then do:

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

Almost! Note that the  $x$  and  $y$  axis labelling is 'correct' and particularly the  $y$ -axis, where latitude goes 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 up-side-down. We can actually do this in the same line as we plot:

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

Phew! (Figure Figure 3.6.)

The final complication is that the data points in the gridded dataset (matrix `etopoldeg`), 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<sup>29</sup>, our vectors that we'd pass into `meshgrid`, would be:

```
» x = [0.5:359.5];
» y = [-89.5:89.5];
```

and hence:

```
» [lon lat] = meshgrid(x,y);
```

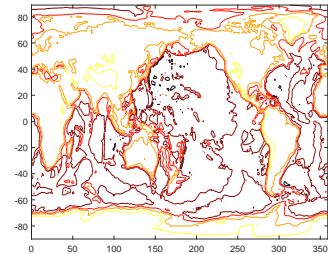


Figure 3.6: Usage of `contour` but with `lon/lat` values created by `meshgrid` function and passed in (and with the `hot colormap` (giving dark/brown colors as deep ocean, and light/white as high altitude).

<sup>29</sup> Don't worry about this for now – grids will be covered more in subsequent 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!). To begin with, pass in just the data matrix (of monthly temperature) when calling the `contour` (or `contourf`) 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);
```

Place this code somewhere before the loop starts in your program.

Now use the longitude and latitude values matrices, in conjunction with `contour(f)`, to plot the global temperature distributions 'properly', e.g.

```
contour(lonv,latv,temp);
```

or if you prefer:

```
contour(lonv(:,:),latv(:,:),temp(:,:));
```

and it helps you remember which variables are arrays.

Try plotting just one plot first (e.g. by adding a breakpoint at the end of the loop, i.e. the line with `end`), 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 if you are into that sort of thing) 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(:,:, )-273.15);
```

would convert to °C as it plotted the data.

You can also (assuming you converted to units of °C) set the plotting temperature limits and contouring consistent between months and with greater color interval resolution by adding the following line (before the loop starts):

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

and then to the contour(... ) (or contourf(... ) ) function, add to the end of the list of passed parameters – v, e.g.

```
contourf(lon(:,:, ), lat(:,:, ), temp(:,:, ), v);
```

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).

---

Lastly – for any (or all) of the Examples above, you could add the continental outline to the plot. Remember, to use hold on in order to overlay the continental outline on top of the contour map without replacing it in the Figure window.

It should be obvious that plotting the continental outline might be something you want to use more than once. Sections of code that might get used multiple times are commonly placed in a file (or special section of the file) of their own and called from the main program that needs it. For example, you could place the entire continental outline plotting code, including loading in the data, in an m-file and make it a function – in this case, taking no parameters as input, and return no output.<sup>30</sup> In the example of the looped animation, in the sequence of code (within the loop), you will need to call your continental outline plotting function just after you have plotted the contour (or bitmap) plotting function.

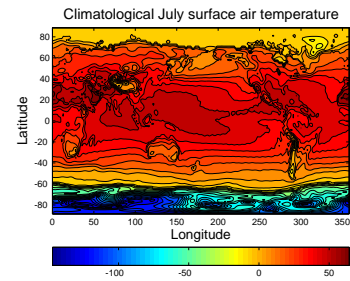


Figure 3.7: Example contour plot including meshgrid-generated lon/lat values. Result of contourf(lon, lat, temp7, 30), where the data file was temp7.tsv, with some embellishments.

<sup>30</sup> Make sure that you do not open a figure window with the figure command with the function, or you will not get a continental output overlay on your plot, but rather a separate Figure window with just the continental outline on.

#### The MATLAB Mapping toolbox

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.



### 3.3 *Further data processing*

This section contains a selection of further simple techniques for doing useful stuff with data, as well as for better graphing.

118 str = 'do you like bananas?'

### 3.3.1 find!

So – a single **MATLAB** function gets its own sub-section, all to itself. Either it's really powerful and useful, or I am running out of ideas for the text<sup>31</sup>.

MATLAB [grey]

`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.

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.<sup>32</sup>

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];
```

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 (in the data array, A).

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

Now, we use `find` to evaluate where in array A (here: a vector) the element with a value of `max(A)` (equal to `c`) occurs. The following should accomplish this:

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

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`, remembering that `find` return an array index (or indices), 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):

<sup>31</sup> 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).

Actually, `find` returns the indices of the non-zero elements in the array and if the array is a vector, what it does is simple. For a matrix, **MATLAB** counts the elements sequentially, starting at the 1st row and 1st column, and working down the first column, rather than provide the (row,column) for indexing format you are used to. Hence where the '*linear indices*' bit comes in.

Furthermore, 'non-zero' indices is really just code-word for 'true'. So you are asking where the true values occur in X. If X is the answer to a *logical* or *relational* operation, then `find` tells you the indices of the elements that are true.

For example, `3 > [5 3 1]` equates to `[0 0 1]`, i.e. only the first element in the vector `[5 3 1]` is less than 3. Hence:

```
find(3 > [5 3 1])
```

first evaluates the relational operation and generates a vector of true and false values, and then `find` tells you the index (or indices) where the true values occur (here, `ans = 3`).

#### min max

Return the minimum and maximum, respectively, values in an array. e.g.

```
min([4 8 3 1])
```

will return a value of 1.

<sup>32</sup> I hope so ... check back earlier (or slightly later) in the course on `max`.

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

```
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 that 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* 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 1s and 0s 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 =
```

```
120 str = 'do you like bananas?'
```

```
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<sup>33</sup> 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) = []`).

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

<sup>33</sup> Use the command `length` or view in the Workspace Window.

---

FOR AN EXAMPLE OF DATA-FILTERING – dig out the paleo-proxy (`paleo_CO2_data.txt`) atmospheric CO<sub>2</sub> data you downloaded earlier. 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<sub>2</sub> 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 that no zero value appear anywhere, 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<sub>2</sub> column of the data with a NaN. The compact version of the command you need is:

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

#### 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 NaNs. The most common/usage in this context is:

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

which will (according to `help`) "generate 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."

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.

```
» 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 (row indices) in the 2nd column of `data` for which the value is equivalent (`==`) to zero. Replace the CO<sub>2</sub> value in all these rows (which is originally zero) 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.<sup>34</sup> Breaking it down, this is similar to before in that you start by identifying the row numbers of where 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<sup>35</sup>, 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<sub>2</sub> proxy data value less than zero(!!!) We can filter this out, just as for zeros. I'll leave this as an exercise for you<sup>36</sup> ... The plot should end up looking like Figure 3.8. As another lesson-ette, given that the circles are insanely large ... try plotting this with proportionally smaller circles<sup>37</sup>.

Conversely, having inserted NaNs into an array, or having ended up with NaNs in an array for other reasons, you can also search for (`find`) the NaNs. The first thing to note in looking for NaNs, is you cannot test for a NaN with a simple equality operator:

```
» a=NaN;
» a==NaN
ans =
    0
```

which ... is odd. Having assigned a NaN to variable `a`, **MATLAB** is apparently telling you that the value of `a` is not equivalent to NaN. Really unhelpful. In fairness, `a` does not have a value and is Not A Number and hence **MATLAB** cannot determine whether or not it is equal to another Not A Number. Better would have been for **MATLAB** to give you an error ... still, it is what it is.

<sup>34</sup> First: Re-load the paleo-proxy atmospheric CO<sub>2</sub> data so that you can have another go at filtering it.

<sup>35</sup> Or:  
» `size(data)`

<sup>36</sup> But you might e.g. use `<=`.

<sup>37</sup> HINT: you are going to want to apply a scaling factor to the vector you passed as the point size data.

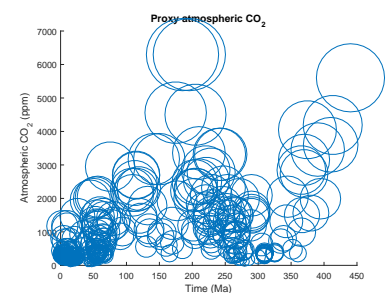


Figure 3.8: Proxy reconstructed past variability in atmospheric CO<sub>2</sub> (scatter plot).

```
122 str = 'do you like bananas?'
```

To try and make amends, **MATLAB** provides a *function* to determine whether or not something is Not A Number – `isnan`. This returns *true* (1) if the passed variable is Not A Number, and *false* (0) if not, e.g.

```
» isnan(a)
ans =
    1
```

whereas:

```
» isnan(99)
ans =
    0
```

because 99 is a number (integer) and not a Not A Number.

---

OPTIONAL ... In the CO<sub>2</sub> 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<sub>2</sub> 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<sub>2</sub> difference, *relative* to the mean, you are going to have to divide by the mean value (the values in column #2 of the array). So first off – if any of the mean values are zero, it is all going to go pear-shaped (dividing by zero ...). 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<sub>2</sub> proxy estimates. We need to get rid of these. Follow the steps as before (i.e. identifying zero or negative values and either removing the corresponding rows in the array, or setting the values to a NaN). 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

#### `isnan`

'`isnan(A)` returns an array the same size as `A` containing logical 1 (true) where the elements of `A` are NaNs and logical 0 (false) where they are not.'

Meaning that you can pass any dimension of array (e.g. vector or matrix or 3D), not just a scalar (a single value or 1 × 1 matrix).

(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<sup>38</sup>, but try and work it out for yourself.)

### 3.3.2 Other data filtering

In the example of the observational Riverside temperature data (data file: `temperature_riverside.txt`), it would be nice to also be able to use `find ...` which can indeed determine all the locations at which a NaN occurs<sup>39</sup>, by e.g.:

```
» find(isnan(temperature_riverside))
```

... but it is not obvious what to 'do' with the resulting list of linear indices for each cell containing a missing value (NaN). For instance, you cannot remove a single cell from a  $a \times b$  array for which both  $a$  and  $b$  are  $> 1$  (because an array must contain the same number of elements in each row (and the same number of elements in each column)).

What you need, is a way of automatically removing each and every row in which one (or more) NaNs appear. There are two obvious approaches:

#### 1. use a loop

We could go through the array, row by row, and for every row in which a NaN occurs, remove that row. The code (in a new *script* file) would look something like:

```
%%% script to load data, remove NaN-containing rows
% load dataset
data=load('temperature_riverside.txt','ascii');
% determine number of rows of data
n_max = length(data);
% initialize row count to the first row
n = 1;
% loop through all rows
while (n <= n_max),
    found_nans = find(isnan(data(n,:)));
    if (~isempty(found_nans));
        % remove row
        data(n,:) = [];
        % remember to update total number of rows!!!
        n_max = n_max - 1;
        % NOTE: don't update row count
    else
        % move on to next row
        n = n+1;
    end
end
end
```

<sup>38</sup> In this possible solution – all rows in the array `data`, with mean CO<sub>2</sub> 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')
```

<sup>39</sup> Using `isnan`.

124 str = 'do you like bananas?'

## 2. cheat!

(not really)

There is a relatively new **MATLAB** function that achieves just this: `rmmissing`. Slightly simpler code that does the same job would then look like:

```
%% script to load data, remove NaN-containing rows
% load dataset
data=load('temperature_riverside.txt','ascii');
% remove problem rows!!!
data = rmmissing(data);
```

### 3.3.3 Basic (pretend) 'stats'

We are not going to delve into complex stats here. A variety of stats related functions are included in the **MATLAB Statistics Toolbox**. We'll stick to some simple functions included as standard in the basic package. Useful basic stats-related functions include:

- `min` – the minimum value of ...
- `max` – the maximum value of ...
- `sum` – the sum of a vector of numbers.
- `mean` – the mean of a vector of numbers.
- `std` – the standard deviations of a vector of numbers.
- `var` – the variance of a vector of numbers.
- `median` – the median value of ...
- `mode` – the mode value of ...

For instance, consider vector `A` (integers from 1 to 9, inclusive):

```
» A = [1:9];
```

Try out all of the above functions on the vector. Most of the values you can pretty much guess. The `mode` of the vector is perhaps the only one where it comes up oddly, because the mode of a set of values is defined as the most popular value, yet you have created only one of each value in the vector. So all values are in theory equally the most frequent and **MATLAB** simply returns the first.

So try adjusting the vector, adding an additional '7' at the end:

```
» B = [A 7];
```

Now what is the mode value of the vector `B`?

Sometimes you have the situation where you have one or more NaNs in the data. For example:

```
» C = [A NaN];
```

`rmmissing` – 'Remove rows or columns with missing entries'.

In the simplest usage:

```
B = rmmissing(A);
```

Removes rows containing missing data elements from array `A`, assigning the results to array `B`.

MATLAB defines missing data as:

- NaN - for number arrays
- <missing> - for string arrays
- blank character [ ' ' ] - for character arrays
- empty character " " - for cell arrays

(see **help** for further information and examples)



Now try out some of the same functions on vector *C*. What happens? Why **MATLAB** does this and does not simply ignore NaNs, is anybody's guess. I mean, what application could you possibly have where when you ask for the mean of a vector, you are hoping to be told 'NaN'? There are solutions.

1. Firstly – you could use `find`, to find and remove NaNs from data. So if you have data that includes NaNs, you could simply filter them out prior to processing the data. The function for determining whether or not an array element is a NaN, is `isnan` (see earlier Box).
2. Or you could create a loop and test each element in turn as to whether or not it is a NaN (again, using the `isnan` function).
3. The **MATLAB** functions listed above, all (most?) have an additional optional parameter (see Box) that allows you to direct **MATLAB** to ignore NaNs in the data.
4. Lastly, in the **Statistics Toolbox**, there are variants of all (most?) of these functions that automatically ignore NaNs, such as `nanmean` (the NaN-ignoring variant of `mean`)

Try out each of these solutions, applying them to the vector *C* (or a NaN-containing vector of your choice).

All of these functions can also be used on 2D arrays (matrices) ... with care. Consider the matrix:

```
» D = [1 2 3; 4 5 6; 7 8 9];
```

(which has the same elements as *A*, just in a different configuration).

What happens when you ask for `mean(D)`? As per help (and the Box): '*If A is a matrix, then mean(A) returns a row vector containing the mean of each column.*' So `mean(D)` is returning the mean of [1 4 7], [2 5 8], and [3 6 9]. Try *transposing* the matrix and then using the *mean function*. You should see that you now get the mean of the individual rows (rather than columns) of matrix *D*:

```
» mean(D')
ans =
    2    5    8
```

This goes for `sum` and all (most?) the rest of the functions.

If you need the total sum of all the elements in a matrix, or mean of all the elements in a matrix, you can simply *nest* the *functions*:

```
» sum(sum(D))
```

or if you prefer breaking things down into sperate steps:

```
» E = sum(D); » sum(E)
```

However, note that `std(std(D))` is not the standard deviation of all elements in the matrix *D*. Why?

### 3.3.4 Some useful data manipulations techniques

This (failure to directly obtain the standard deviation of all the elements in a matrix) brings us to array manipulation, which will prove useful in other contexts, such as graphing and particularly scatter plots. In the previous example, `std(std(D))` fails to give us what we want. Ideally, we would like to have all the elements of `D` re-configured so that they were in a single vector format, and we could just write e.g. `std(D)` and get a single, complete answer.

**MATLAB** provides the function `reshape`<sup>40</sup> for the express purpose of re-configuring the shape of an *array*, such as turning a *matrix* into a *vector*, or *vice versa*. For instance, given the 3x3 matrix `D` in the previous example:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

how do we turn this into a 9x1 column vector?

$$\begin{pmatrix} 1 \\ 4 \\ 7 \\ 2 \\ 5 \\ 8 \\ 3 \\ 6 \\ 9 \end{pmatrix}$$

You can use `reshape` in 2 different ways:

1. Firstly, you can explicitly specify the new array shape you want.<sup>41</sup> e.g.

```
Dvector = reshape(D, [9,1]);
```

2. Alternatively, if you know you want a single column vector and cannot be bothered to work out how many rows you need, **MATLAB** will kindly pick up the slack via a slightly different usage of `reshape`:

```
Dvector = reshape(D, [],1);
```

Here you are specifying one column, but 'whatever' (`[]`) rows.

(Having created a vector containing all the numbers, you can now find the standard deviation: `std(Dvector)`.)

Obviously, if you want a row, rather than a column vector – either transpose the column vector to row vector shape, or specify the format of a row vector in the first place when using `reshape`:

#### **reshape**

Use `reshape` to transform data in an array of one shape (i.e. configurations of rows and columns), into another. **MATLAB** help is OK on this and for the main usage of the function, says:

*'B = reshape(A,sz) reshapes A using the size vector, sz, to define size(B). For example, reshape(A,[2,3]) reshapes A into a 2-by-3 matrix.'*

In this usage you need to specify the rows and columns of the resulting array.

NOTE that the array you turn it into to, can have a single row, or a single column (and hence be a vector), but you need to specify this with a 1.

Also note that the total number of elements in the array must be conserved, so if you turn an  $n \times m$  array into a  $p \times p$  array, then it must be true that:

$n \times m = o \times p$  There is also a convenient second usage, that will attempt to automatically determine the row or columns needed to make  $n \times m = o \times p$  true, given either  $o$  or  $p$ . For example:

```
B = reshape(A, 2, [])
```

in the previous example will automatically determine that 3 columns are needed. Conversely,

```
B = reshape(A, [], 3)
```

will determine that 2 rows are required to meet the  $n \times m = o \times p$  criteria.

This usage is particularly convenient for making vectors, e.g.:

```
B = reshape(A, [], 1)
```

<sup>40</sup> See `help` and `Box`

<sup>41</sup> Obviously, the total number of elements in the array must be conserved.

```
Dvector = reshape(D,[1,9]);
```

Lastly, you might notice how **MATLAB** reads the elements from **D** before creating the new array shape – elements are read from the top of the first column downwards, before moving to column #2. Hence why the order of numbers is 1 4 7 2 5 8 ...

What, instead, if you wanted a vector with the ordering:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{pmatrix}$$

?

ANS: simply *transpose* the matrix before you reshape:

```
Dvector = reshape(D',[1,9])
```

(or `Dvector = reshape( transpose(D) , [1,9] )`).

### 3.3.5 Data interpolation

Interpolation? What is it and why would you do it? We'll answer this via an example.

First download the ice-core dataset of atmospheric CO<sub>2</sub> over the past 800,000 years, recovered from the Dome C site on Antarctica – filename `icecore_co2.txt` on the course webpage. Start by plotting it (your choice of **MATLAB** plotting function) to see what you are dealing with.<sup>42</sup>

So what if we wanted to know the average (mean) value of atmospheric CO<sub>2</sub> over the last full glacial cycle, i.e. between now (age zero) and the end of the previous interglacial period, about 115,000 years ago.

So firstly, you might use your most excellent **MATLAB** skills to extract all the data corresponding to this specific interval – i.e. all the ages (and corresponding CO<sub>2</sub> values, between zero and 115,000 years, or rather, less than or equal to 115,000 years. You should know this requires the `find` function, and that the range of indices is given by (assuming the data array you loaded in is called `co2`):

```
» a=find(co2(:,1)<=115000)
```

which simply says: take all the elements in the 1st column of the array `co2` (`co2(:,1)`), and find the indices of all the elements with a

<sup>42</sup> First column / x-axis values are age, in years, and 2nd column / y-axis values are CO<sub>2</sub> concentration, in units of ppm.

```
128 str = 'do you like bananas?'
```

value equal or less than 115,000. To select just the first 115,000 years of data in `co2` is then just a matter of:

```
» co2new=co2(a,:)
```

and check that this does indeed give you the correct portion of data and has assigned it to the array `co2new`. (Maybe plot to confirm.)

It is worth pausing at this point – this is a common, and powerful, usage of `find`, and of *indexing*, and you should be sure you understand it before moving on. What this line is saying is: take both columns of the array `co2` – select all the elements (rows) defined by the vector `a`, and assign the result to `co2new`.

OK, so we are progressing well towards answering the question – the mean  $CO_2$  value over the last glacial cycle (last 115,000 years). In fact – try answering that now (using `mean`). You should end up with a value of 245ppm.<sup>43</sup> The question is – do you ‘believe’ it? Look at the plot – do you think that value is representative of the average?

To make the problem more obvious – repeat the above exercise, but now consider only the past 40,000 years. From the plot, high, interglacial  $CO_2$  values characterise only the last 10,000 years or so, with a transition over 5,000 years or so before that. From 15,000 years and back to 40,000,  $CO_2$  is clearly bumping along its lowest values. What would you guess the mean  $CO_2$  value is? Now try it. I get 249ppm  $CO_2$ . Does that look correct to you, across the past 40,000 years?

If you were previously using `plot` to plot the data, now try `scatter`. It should be much more obvious what is going on now – you have very uneven data sampling in time – the bulk of the data is from the last 10,000 years or so, and there are very few data points older than about 22,000 years. When **MATLAB** calculates a mean, it is of the data points, and an uneven data sampling will give a biased, unrepresentative value.

We need to interpolate the data – place it on an evenly sampled-in-time basis. The **MATLAB** function to interpolate vector (1D) data is `interp1` (see `help/Box`).

The first thing we need, to use `interp1`, is a vector of points in time, that we are going to interpolate our data on to. As a rule, the vector should ideally not extend in value beyond the minimum and maximum values of the original axis, but we’ll ignore this for now. We might pick ... 1,000 years as a simple sampling interval, and so to create this new axis vector, we would write:

```
» xi=[0:1000:40000];
```

assuming we stick with the 0-40,000 year interval. The `interp1` function requires that you pass this vector, along with the original

<sup>43</sup> Note that **MATLAB** will report the value in a scientific notation with a power (here  $10^2$ ).

#### `interp1`

```
yi = interp1(x,y,xi)
```

will interpolate the y-axis values located at x-axis points given by the vector `x`, onto the x-axis points given by vector `xi`. The resulting interpolated y-values are assigned back to `yi`.

By default the interpolation methods used is linear. For a different interpolation method, use the variant of the function:

```
yi =  
interp1(x,y,xi,method)
```

where `method` is one of:

```
'nearest', 'linear', 'spline',  
'cubic' ... (for a fuller list, see  
help).
```

To extrapolate outside of the domain spanned by the (original) x-axis vector `x`, specify:

```
yi =  
interp1(x,y,xi,method,'extrap')
```

time (x-axis) vector, and the original data (y-axis) vector, and will give you a new data vector, with values corresponding to the time points defined by `xi`. Like this:

```
» yi=interp1(co22(:,1),co22(:,2),xi);
```

If you prefer to break things down<sup>44</sup> so as the process is clearer, maybe first extract and create the original data, x-axis (time) vector:

```
xold=co22(:,1);
```

and then the y-axis (data) vector:

```
yold=co22(:,2);
```

and then do the interpolation:

```
y1=interp1(xold,yold,x1);
```

Either way, now scatter-plotting the interpolated data:

```
» scatter(x1,y1);
```

should result in an obviously evenly-spaced data plot.

We could now use `mean`, except if you were paying attention, because we extrapolated outside of the range of the extracted data into the array `co2new`. But you know how to handle this situation, i.e. removing the offending NaN rows, or use `nanmean` if you have access to the required **MATLAB** toolbox.

Or, you could re-do the interpolation, but interpolate from the full, original data array, which you know extends way past 40,000 years. And ... specify the very first time point as 1,000 years rather than zero. e.g.

```
x1=[1000:1000:40000];
y1=interp1(co2(:,1),co2(:,2),x1);
```

Well ... it doesn't work, which is sort of pretty 'real world' problem-  
esk. The issue is that there is a duplicate year – i.e. 2  $CO_2$  values with the same year.<sup>45</sup> How to find them? Well, you saw earlier the function `mode`, which return the most popular value in an array. If we do:

```
» mode(co2(:,1))
ans =
    409383
```

Ha ha, so the year 409,383 is duplicated.<sup>46</sup> How to find this ...

```
» find(co2(:,1)==409383);
```

or if you prefer (and neater):

<sup>44</sup> And then you might also make the variable names REALLY explicit, and have `xold`, `yold`, `xnew`, `ynew` or something.

<sup>45</sup> The **MATLAB** interpolation function requires a strictly monotonically increasing (with no duplicates) old and new x-axis vector.

<sup>46</sup> The absence of duplicated year values, or rather, there only being one of each individual number, would result in **MATLAB** returning the very first value in the vector. The only confusion here would be if the very first value was itself duplicated ...

```
130 str = 'do you like bananas?'
```

```
» » rows=find(co2(:,1)==mode(co2(:,1)))
ans =
    531
    532
```

giving us the row numbers. We could be clever, and create a single entry for this year, with the  $CO_2$  value formed of the mean of the duplicate entries:

```
» co2mean=mean(co2(rows,2));
```

... replacing the first  $CO_2$  value ...

```
» co2(rows(1),2)=co2mean;
```

... and then delete the second.

```
» co2(rows(2),:)=[];
```

Phew! Now try the interpolation again. Plot (`scatter` and/or `plot`). Find the mean  $CO_2$  value over 0-40,000 (technically, now we have restricted the data to 1,000-40,000). Does this seem more reasonable? Repeat for 0-115,000 years (or 1,000-115,000 years). Also try out carrying out an interpolation with closer spacing, say 500 or 100 years.

Finally, if you read the (`Box` or `help`) details about the function `interp1`, the more recent **MATLAB** version releases enable you to extrapolate outside of the data domain. So instead of having to restrict the `xi` vector to starting at year 1,000, you can start at year zero:

```
» x1=[0:1000:40000];
» y1=interp1(co2(:,1),co2(:,2),x,'linear','extrap');
```

### 3.4 *Even nicer graphing and graphics*

There are a bunch of simple **MATLAB** drawing and text placement commands that can help improve the look and feel of scientific plots, or even replace the provided plotting functions (i.e. you can create your own bespoke plotting functions). There are also a variety of options for altering the axes, axes tick-marks, axes tick-mark labels, etc that can be useful.

```
132 str = 'do you like bananas?'
```

### 3.4.1 Drawing lines and shapes

We'll start with some simply line and shape drawing. At the command line – open a new figure window (`> figure;`). Then before anything else, do a `> hold on;`.

When **MATLAB** draws lines and shapes and places text, it needs to know the coordinates of where to place things. It is not possible (I think) to draw directly in the figure window – **MATLAB** needs a frame to put things in, and the easiest way to do this is to create a set of axes. So having opened a new figure window, set the range of the x- and y-axes <sup>47</sup>:

```
> axis([0 100 0 100]);
```

The resulting figure is really not very exciting ... and should look like Figure 3.9.

---

There are 2 ways to draw a line (examples assuming the previous Figure window remains open with `hold on ...`):

#### 1. plot

Are you recall, `plot` will plot a sequence of (x,y) points, and by default, join them up with a line. If we wanted a diagonal line, from the original to the mid-point of the plot area, we could invent a pair of vectors to define the two points we need – at (0,0) and (50,50):

```
x1 = [0 50];  
y1 = [0 50];
```

and then plot the resulting points as a `plot` plot:

```
plot(x1,y1);
```

You should now see something like Figure 3.10.<sup>48</sup>

#### 2. line

**MATLAB** provide a specific command for drawing lines ... `line`. In its simplest usage, it is a little like `plot`, except taking only a single pair of x- and y-coordinate values.

To use `line` to draw a 2nd line segment, starting at (50,50) and terminating at (100,0), we create another pair of vectors to define these points:

```
x2 = [50 100];  
y2 = [50 0];
```

and then draw it:

```
line(x2,y2);
```

as shown in Figure 3.11.

<sup>47</sup> Here taking the example range of 0-100 on both axes.

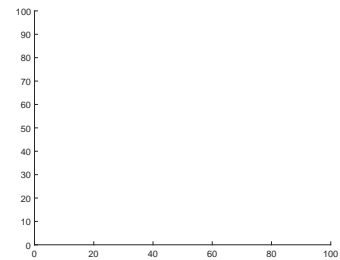


Figure 3.9: Figure window with axes.

<sup>48</sup> If you find that the plot area has been re-sized such that the x- and y-axes now both go from 0-50, then you have forgotten to do a `hold on`.

#### `line`

To draw a simple (single) line on a graphic:

```
> line([x1 x2],[y1  
y2])
```

where `x1` and `x2` are the x-coordinates of the start and end position of the line, and `y1` and `y2` are the corresponding y-coordinate values.



Obviously, both line segments could be drawn using `plot`, or both with `line`.

If you are just drawing, rather than annotating a plot with axes, then you might want to turn off, or hide, the axes tick marks and tick labels. To do this, we first need to find the special **MATLAB** ID of the axes, which helpfully, because you only have one set of axes and have just been using them, is the 'current axis'. To do this, we use the `gca` function, which returns the *handle* (ID) of the axes:

```
» h = gca;
```

Having got the axis *handle*<sup>49</sup>, we can now 'set' the properties of the axes, using `set`:

```
» set(h,'XTick',[],'XTickLabel','');
```

What this does is to tell **MATLAB**: take the graphics object with the ID contained in variable `h` (which we just retrieved via the `gca` function), and `set` (which is why we use the command `set ...`) the following properties:

- `'XTick', []` – set the number and position of tick marks on the x-axis, to the contents of the empty vector `[]`.
- `'XTickLabel', ''` – set the labels applied to the tick-marks, to `''` (i.e. no text).

Actually, in this example, the 2nd graphics parameter set (the labels) is sort of redundant, as there are no tick-marks in the first place ...

To see how different combinations of settings pan out, try:

```
» set(h,'XTick',[0 50 100],'XTickLabel','');
```

(3 small inwards ticks, no labels)

```
» set(h,'XTick',[0 50 100],'XTickLabel','cat');
```

:o)

and/or:

```
» set(h,'XTick',[0 50 100],'XTickLabel',{'cat',
'dog', 'rabbit'});
```

where `{'cat', 'dog', 'rabbit'}` is actually a 3 string *cell array*.

All this insanity should be looking like Figure 3.12 (if we also remove the y-axis ticks and labels<sup>50</sup>).

An alternative way to create a figure to draw on, without having to remove the axes ticks and labels etc etc, is to create the axes as invisible. To try this – first create a new figure window.<sup>51</sup>

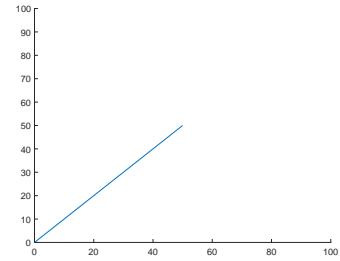


Figure 3.10: Figure window with single line segment (via `plot`).

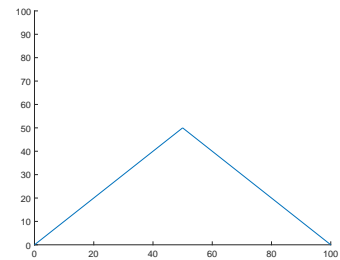


Figure 3.11: Figure window with a second line segment (via `line`).

<sup>49</sup> It is worth omitting the `;` in order to see the properties associated with the axes, and in fact, it is worth clicking on Show [all properties](#) to see a list of everything that can be edited and adjusted.

ALL these can be changed if you ever want!!!

<sup>50</sup> It is sufficient just to type:

```
» set(h,'YTick',[]);
```

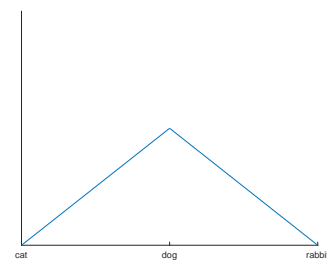


Figure 3.12: (no comment).

<sup>51</sup> If you find yourself drowning in figure windows, remember that `close` closes the current window, and `close all` closes all of them.

```
134 str = 'do you like bananas?'
```

```
» h = axes('Position',[0 0 1 1],'Visible','off');
```

Here, 'Position',[0 0 1 1] specifies that the axes area should fill the window, and 'Visible','off' says to make the axes invisible. (We keep a copy of the *handle*, *h*, just in case we need it later.) And set the axes to the same 0-100 limits as before:

```
» axis([0 100 0 100]);
```

Then hold on.<sup>52</sup>

Now try drawing some lines (remembering the 0-100 axes limits when making up (x,y) co-ordinates).

<sup>52</sup> For some reason ... you need to do hold on only after creating axes frame ...

---

The command `set` can be used in the context of any (I think?) graphics object, i.e. any individual component part of a final plot such as the axis line itself, the axis ticks, plotted lines and points, etc. For example, in creating the line segment previously:

```
» h = line(x2,y2);
```

you could store a copy of the *handle* of that line segment – here it is being assigned to the *variable* *h*. With this, you can now change the properties of the line (after you have drawn it). such as by:

- » `set(h,'LineWidth',2.0);`

will change the line width to 2.0 (points).

- » `set(h,'Color',[1 0 0]);`

will turn the line red, using the *RGB* (red-green-blue) notation: [1 0 0]<sup>53</sup>.

- » `set(h,'LineStyle',':');`

will make the line dotted.

<sup>53</sup> Alternatively:

```
set(h,'Color','r');
```

Note that here, you are setting line properties after you have created the graphics, whereas earlier in eg. using `plot` to graph data, you specified the properties as you draw the lines. Both ways are valid, but being able to change properties later and after plotting, gives you greater flexibility. Note that after plotting a graphic, you can also edit and adjust its properties in the Figure window itself (via its GUI)

---

Using a handle also now enables you to complete an earlier plot. For the proxy CO<sub>2</sub> data where you color-coded the points and added a `colorbar`, there was not actually 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, you would add to your script file (or type at the command line):

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

The revised plot should then end up looking something like Figure 3.13 in which you can see the high relative uncertainty (bright colors) prevail at low CO<sub>2</sub> 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.

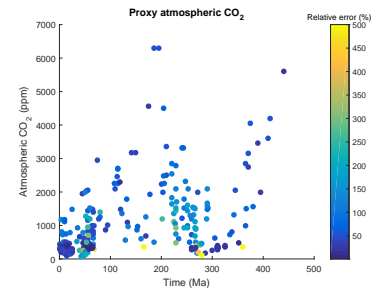


Figure 3.13: Proxy reconstructed past variability in atmospheric CO<sub>2</sub> (scatter plot).

An obvious use for drawing lines on plots, is to annotate them. e.g. placing a text label (we'll see shortly), with a line pointing from the text to a specific feature. You can do with with a simple line and hence the `line` command.

It would be more handy and in fact common, to include an arrow head to make clear that the line is pointing to something. This can in theory be done by drawing 2 more, shorter lines, but is no fun at all<sup>54</sup>. **MATLAB** hence provides the function `quiver`. `quiver` is commonly used for plotting fields of arrows, but can equally be used to create a single arrow – much like earlier you used `plot` to draw just a single pair of joined up points and hence a line. However, rather than take a pair of (x,y) points – one for the start and one for the end, of the line, `quiver` takes an (x,y) location for the start of the arrow, and then the length in the x and y directions.

Consistent with the previous example, we were starting the line segment at (0,0), and then extending the line to (50,50). The length vector in this case is also [50 50]. So, given the specific syntax and input parameter format required by **MATLAB** for this function, we would write:

```
» quiver(0,0,50,50,0);
```

Here – the last, 5th parameter (0), tells **MATLAB** not to auto-scale the arrow.<sup>55</sup>

For 2D shapes, you can draw rectangles using `patch`. This takes as parameter input, a vector of x-coordinate positions, then a vector

<sup>54</sup> True fact – I have tried it :(

<sup>55</sup> If it your arrow hard to make out – try creating a new figure window. You can also use `cls` to clear all the graphics in the window (i.e. and not have to re-generate the figure window.

of y-coordinate positions, and as a 3rd parameter, the color for the object.

So in our previous example, with the x- and y-axes going from 0-100, say we want to draw a square in the middle, 20 units on each side. We could create our vector of x-axis coordinates as such:

```
» X1 = [40 40 60 60];
```

and for the y-axis ... some care is needed and often it might be helpful to sketch out the coordinate pairs and positions on a piece of paper:

```
» Y1 = [40 60 60 40];
```

And then:

```
» patch(X1,Y1,'r');
```

as shown in Figure 3.14. Note that in this (X1, Y1) point notation, we are telling **MATLAB** to plot a shape with vertices at:

```
(40,40), (40,60), (60,60), (60,40)
```

Also note that the order of the (x,y) pairs, matters, as **MATLAB** draws the line segments between the vertices in the order that they are given to **MATLAB**. For the same (x,y) pairs, try creating X1 and Y1 vectors with the pairs in a different order and see what happens.

Slightly changing the values of the (x,y) pairs can also give you a diamond (-ish):

```
» X2 = [40 50 60 50];
» Y2 = [50 60 50 40];
» patch(X2,Y2,'c');
```

as shown in Figure 3.15.

`patch` is in fact much more flexible than I have shown so far, and in fact, will draw any polygon. Consider this sort of slightly random series of x and y coordinates:

```
» X3 = [20 40 60 80 60 40];
» Y3 = [50 60 50 60 40 30];
» patch(X3,Y3,'g');
```

gives Figure 3.16.

Try designing/playing about with different shapes. Perhaps sketch them out on paper first and list down the coordinates before telling **MATLAB**.



Figure 3.14: Square.



Figure 3.15: Alt square.

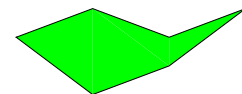


Figure 3.16: Random polygon.

---

If you have the **MATLAB** Image Processing Toolbox installed, then you can use the command `viscircles` to draw circles.

If not – a crude but sometimes effective alternative, is to `scatter` plot a single point ((x,y) location), and set a large size value for the circle. For example:

```
» scatter(50,50,1000);
```

or filled:

```
» scatter(50,50,1000,'filled');
```

### 3.4.2 Colors

You are already familiar with setting colors for lines, with the notation: 'r', or 'b' (for red, blue, respectively). This is nice and simple and so totally fabulous ... except there are a limited number of colors available in this notation (see Box).

Hence there is an alternative that enables a more exact specification of color. In this particular scheme – red-green-blue, abbreviated to *RGB*, you set the intensity of red, green, and blue, on a scale of 0 to 1. And supply this in a vector format to **MATLAB**. For example:

- [0 0 0] – zero intensity of all of R, G, B => black.
- [1 1 1] – 100% intensity of all of R, G, B => white.
- [1 0 0] – 100% intensity R, none for G and B => red.
- [0.5 0.5 0.5] – 50% intensity of all of R, G, B => grey.
- [0.5 1.0 0.5] – light green.

Play around with some RGB value combinations, plotting shapes, or filled circles, e.g.

```
» scatter(50,50,1000,[0.25 0.75 0.25],'filled');
```

A rendition of the RGB color scale is shown for reference in Figure 3.17.

### 3.4.3 Placing and making text 'nice'

There is not much to placing text and specifying its properties. The **MATLAB** command for writing a string to a figure window, is `text`. That's it! (see Boxes)

For instance, you could write:

```
» text(25,25,'bananas');
```

and the text `bananas` will appear at location (25,25) on your plot.

Additional parameters can be added to change font, size, etc (see Box), e.g.:

```
» text(25,25,'bananas','FontSize',24,'Color',[0 1 1]);
```

for big light blue bananas.

#### MATLAB quick colors:

- y – yellow
- m – magenta
- c – cyan
- r – red
- g – green
- b – blue
- w – white
- k – black

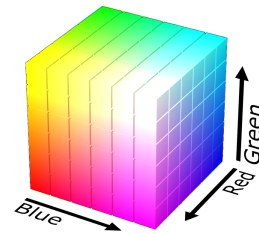


Figure 3.17: RGB scale. By SharkD - Own work, GFDL, <https://commons.wikimedia.org/w/index.php?curid=3375025>

#### text

```
text(X,Y,STRING);
```

will write the string contained in the variable `STRING` (or you can pass the text as a string directly), at location (X,Y).

Note that by default, **MATLAB** aligns the left-hand edge of the text with the X coordinate position, and the mid-point of the string vertically, with the Y coordinate. i.e. the string is *left-aligned* and *centered* vertically.

A variety of additional properties can be set at the time, e.g.

```
text ...
(X,Y,STRING,'FontSize',12);
```

specifies a 12 pt font size. Other common parameter options include:

- 'FontName'
- 'Color'
- 'Rotation'
- 'HorizontalAlignment'
- 'VerticalAlignment'

See **MATLAB** help for more details.

### 3.4.4 Creating color maps

As mentioned earlier – **MATLAB** enables a range of different color scales (*colormaps*) to be used in (esp. contour) plotting and provides around dozen built-in possibilities (see Box).

Taking the earlier example of loading and plotting the global topography data:

```
» data = load('etopoldeg.dat', '-ascii');
» imagesc(data);
```

gives Figure 3.18, and

```
» colormap('hot');
» imagesc(data);
```

gives Figure 3.18.

You can also define your own *colormap*. *Colormaps* are simply a matrix of  $[RGB]$  colors. The most trivial *colormap* would be:

```
» cmap1 = [0 0 0];
» colormap(cmap1);
```

creates and applies a *colormap* containing a single color (black). Try it ... but it is clearly not very useful ...

Better, would be:

```
» cmap2 = [0 0 0; 1 1 1];
» colormap(cmap2);
```

which creates and applies a color scale containing 2 colors - black and white and when used for the topography data, gives Figure 3.20.<sup>56</sup>

You can keep adding colors, e.g.

```
» cmap3 = [0 0 0; 0.5 0.5 0.0; 0.0 0.5 0.5; 1 1 1];
```

but this is a lot of effort to keep adding single additional colors.

What you really want to do, is to define end-member colors, and then tell **MATLAB** to *interpolate* in between these. Recalling back a couple of subsections:

```
» ynew = interp1(xold, yold, xnew);
```

takes the y-values (*yold*) at x-values *xold*, and interpolates onto the x-values defined by the vector *xnew* (and assigns the new y-values to vector *ynew*). For instance, if we have the following crudely spaced data<sup>57</sup>:

#### colormap (2)

As mentioned earlier, **MATLAB** has a number of 'colormaps' built in, which are:

- parula (default)
- jet (old default ... avoid ...!)
- hsv
- hot
- cool
- spring
- summer
- autumn
- winter
- gray
- bone
- copper
- pink
- lines
- colorcube
- prism
- flag

and which you can set by:

```
» colormap NAME
(or colormap(NAME)), e.g.:
» colormap 'hot'
(or colormap('hot'))
```

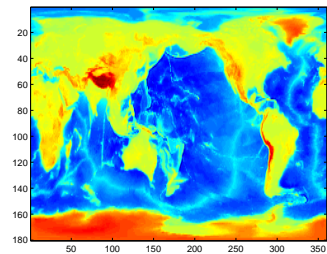


Figure 3.18: Global topography plotted with the default **MATLAB** color scheme.

<sup>56</sup> Remember, *imagesc* plots using the maximum number of colors available, and in this example, the mid value between the deepest place in the ocean and highest point on land, divides the colors into black and white – within specifying a particular scale, this color separation does not occur at zero (sealevel)

<sup>57</sup> In **MATLAB** notation:

```
xold = [0 3 7 13 16 22
30];
yold = [0.2 0.6 0.7 0.3
0.3 0.1 0.0];
```

$$\begin{pmatrix} 0 & 0.2 \\ 3 & 0.6 \\ 7 & 0.7 \\ 13 & 0.3 \\ 16 & 0.3 \\ 22 & 0.1 \\ 30 & 0.0 \end{pmatrix}$$

and we wanted to create an interpolated dataset from 0.0 to 30.0 (in x-axis values) in steps of 1.0, we would first create the new x-axis vector that the data will be interpolated on to:

```
xnew = [0.0:1.0:30.0];
```

and then we would write:

```
» ynew = interp1(xold,yold,xnew,'spline');
```

and obtain the interpolated data as shown in Figure 3.21. (Try this.)

We can do something similar for the *colormaps*. Consider the simple end-member black-to-white white scale:

```
» cmap2 = [0 0 0; 1 1 1];
```

We can write this as points along a vector  $x$  (the axis not representing anything in particular – the number of the color, or simply the normalized distance between the extreme end-member colors), together 3 color vectors (for the separate red, green, and blue component values):

$$xold = \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix}, rold = \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix}, gold = \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix}, bold = \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix}$$

and:

```
» xold = [0.0; 1.0];
» rold = [0; 1];
» gold = [0; 1];
» bold = [0; 1];
```

If we want to create a scale of 11 total (from 0.0 to 1.0 in steps of 0.1) different colors, we can create a new  $x$  vector to interpolate on to:

```
xnew = [0.0:0.1:1.0];
```

and then either interpolate the 3 color vectors separately:

```
rnew = interp1(xold,rold,xnew,'spline');
gnew = interp1(xold,gold,xnew,'spline');
bnew = interp1(xold,bold,xnew,'spline');
```

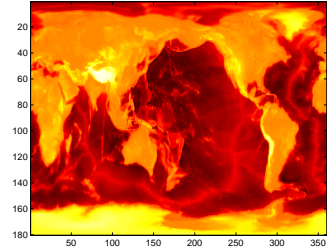


Figure 3.19: Global topography plotted with hot.

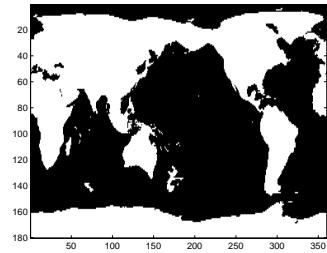


Figure 3.20: Global topography plotted with a basic black+white dual color scheme.

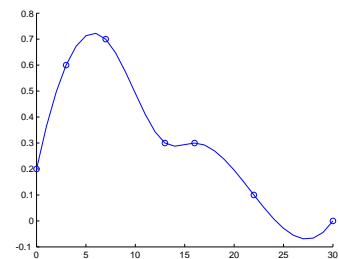


Figure 3.21: Comparison of sparsely sampled data (points) compared with a more finely spaced spline interpolation (solid line). (x-axis and y-axis are both unit-less.)

```
140 str = 'do you like bananas?'
```

or **MATLAB** allows us to interpolate all together if we first combine the sperate vectors:

```
mapold = [rold gold bold];
```

and then:

```
mapnew = interp1(xold,mapold,xnew,'spline');
```

If you now set the new *colormap* ( $\gg$  `colormap(mapnew);`) and re-plot the global topography, you should get Figure 3.22.

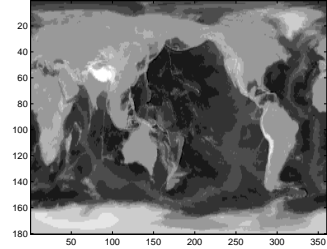


Figure 3.22: Global topography plotted with a user-defined grey-scale.



# 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) <sup>1</sup>.

<sup>1</sup> According to the all-mighty Wikipedia (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."

### Search algorithms

Lets assume that you have a function:

$$y = f(x)$$

There are two common cases that you might want to solve (or approximate):

1. The value of  $x$  such that the value of  $f(x)$  is minimized ( $y \simeq 0$ ).
2. The value of  $x$  such that the value of  $\frac{dy}{dx}$  is minimized (first derivative  $\simeq 0$ ).

Lets further assume that you can place some initial limits on  $x : x_{min} \leq x \leq x_{max}$ .

A good place to start in both examples is to test the mid-point of the limits:  $f(\frac{x_{min}+x_{max}}{2})$  (In some cases you might instead take the log-weighted mean.)

In case #1 and assuming that  $\frac{dy}{dx}$  is positive, if:

$$f(\frac{x_{min}+x_{max}}{2}) > 0$$

you replace  $x_{max}$  with  $\frac{x_{min}+x_{max}}{2}$  (the current tested value of  $x$ ) and if:

$$f(\frac{x_{min}+x_{max}}{2}) < 0$$

you replace  $x_{min}$  with  $\frac{x_{min}+x_{max}}{2}$ .

Keep repeating until the difference  $y$  and zero falls beneath some specified tolerance.

In case #2, you need to test the value of  $f(x)$  infinitesimally away from  $f(\frac{x_{min}+x_{max}}{2})$  to determine whether the gradient is positive or negative (assuming that you do not *a priori* know the derivative function). The idea here is to ensure that the values of  $x_{min}$  and  $x_{max}$  correspond to positive and negative (or negative and positive) gradients. i.e.  $\frac{x_{min}+x_{max}}{2}$  replaces  $x_{min}$  or  $x_{max}$  according to which has the same sign of gradient.

#### 4.1 Nested loops

A helpful device, particularly when dealing with arrays of data in **MATLAB**, is to *nest* loops – i.e. placing one loop inside another one. (So far, you have seen single loops, and single loops with *conditionals* inside, but not nested loops.) A generic code for a nested loop might look like:

```
% loop 1 start
for n=1:10
    % loop 2 start
    for m=1:10
        %% CODE
    end
end
end
```

Here, the value of *n* cycles (loops) from 1 to 10 (i.e. the loop goes around 10 times). Then ... for each value of *n*, the value of *m* also cycles from 1 to 10. The code in the middle of the innermost loop is then executed a total of  $10 \times 10 = 100$  times.

Try this (in a new script file) and confirm that the outer loop cycles around 10 times, and the inner loop ten times for each cycle of the outer loop. (e.g. you might add a `disp` line both within the inner loop, and outside of the inner loop but within the outer loop.)

Why would you do this (what use could it be)?

Imagine you are programming a game of Tic-tac-toe (in fact we will, in a later chapter!). The drawn grid might look like Figure 4.1.<sup>2</sup>

In terms of **MATLAB** and computer programming, we might create a representation of the grid, and assign 0 to unpicked squares, a 1 for where a cross is, and a 2 for where a naught is, as per Figure 5.7 (because we cannot numerically represent an actual cross or circle shape).

To store this information, we could create an *array* in which each location would have a value of 0, 1, or 2, e.g.

$$\begin{pmatrix} 1 & 2 & 0 \\ 1 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

and as per Figure 4.3

OK – ignore the existence of the **MATLAB** *function* `find`, and lets say that you want find the locations of the crosses – '1's in the array code notation. You need to test each an every location in the array (lets call it `tokens`) in turn for whether its contents is a '1' or not. We could do this long-hand ...

```
if ( (tokens(1,1) == 1) || (tokens(2,1) == 1) ||
    (tokens(3,1) == 1) || (tokens(1,2) == 1) || ...
```

<sup>2</sup> In this case, player x has obviously already won. What was naughts thinking???

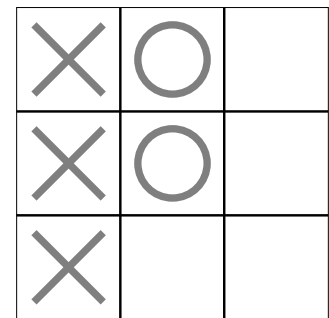


Figure 4.1: Tic-tac-toe game grid.

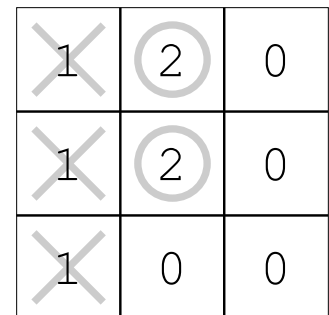


Figure 4.2: Tic-tac-toe game grid with numerical codes overlain.

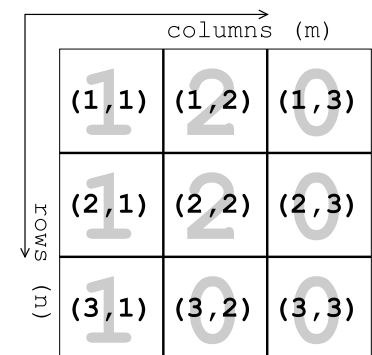


Figure 4.3: Tic-tac-toe game grid – numerical representation.

... but would get desperately tedious pretty quickly.<sup>3</sup> And what if the grid (array) was  $100 \times 100$ ? You could have to have 10,000 tests of an equality in the `if` ...

The idea then is to *loop* through all the locations in the array in turn. And we do this by: For each row (which we search through in a loop), we loop through all the columns. Our code fragment would then loop like:

```
% loop 1 start
for row=1:3
    % loop 2 start
    for column=1:3
        if (tokens(row,column) == 1),
            %% CODE
        end
    end
end
```

This nested loop ensures that you visit each and every array location in turn – working across every column, for each and every row, as per Figure 4.4 and test for the occurrence of a value of 1.

We could also carry out the search of (*rows, columns*) in the opposite order:

```
% loop 1 start
for column=1:3
    % loop 2 start
    for row=1:3
        if (tokens(row,column) == 1),
            ...
        end
    end
end
```

and now loop through each row (inner loop) for each column (outer loop) as illustrated in Figure 4.5. The result is exactly the same. To some extent, which axis direction you choose as the outer loop is a matter of personal preference<sup>4</sup>.

You can also search in the opposite direction (sign), e.g.

```
% loop 1 start
for column=1:3
    % loop 2 start
    for row=3:-1:1
        if (tokens(row,column) == 1),
            %% CODE
        end
    end
end
```

searches across columns, from left-to-right, but rows in the order bottom-to-top. This perhaps looks a little like how you might visualize a search on a (*lon,lat*) grid(?)

<sup>3</sup>Note what is being done here – we are using a series of ORs (`|`) to determine whether any of the array locations (squares) contain a value equal to 1.

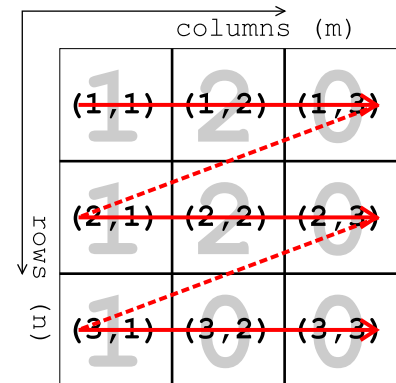


Figure 4.4: Tic-tac-toe game grid – search order: columns then rows.

<sup>4</sup>When arrays are small, the the overall computational expense small.

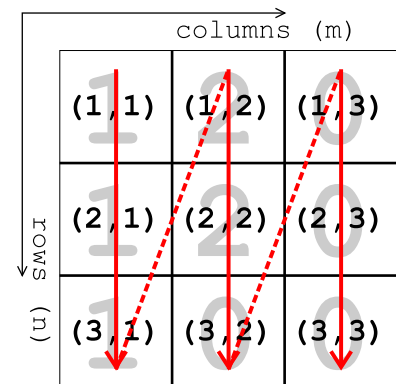


Figure 4.5: Tic-tac-toe game grid – search order: rows then columns.

The concept is the same even for very large grids (where you cannot easily draw a graphical representation to help you).

Nor, do the number of rows and columns have to be the same. For example you might want to access information stored in an array that has a cell location for every day of the year. In this case, you might have 12 columns for the 12 months, and 31 rows so that you can accommodate the number of days in the longest month.<sup>5</sup> In fact, in this example, the inner loop – days – might have a different loop maximum, depending on which month, e.g.

```
% month loop start
for month=1:12
    % determine length of month
    switch month
        case {1, 3, 5, 7, 8, 10, 12}
            day_max = 31;
        case {4, 6, 9, 11}
            day_max = 30;
        case 2
            day_max = 29;
    end
    % day loop start
    for day=1:day_max
        %%% CODE
    end
end
```

Here, the `switch ... case` structure is used to test for which month (number) it is, and set the day-of-the-month loop limit accordingly. Note the use of curly brackets `{ }` in defining the list of elements in the case statement. (A single element does not require curly brackets.)

Try this code (in a new *script* file). Add a `disp` line within the day loop to confirm that the correct number of days is being counted up to each month.

Also – try (in the same *script* file, or create a new one) replacing the `switch ... case` structure, with `if ... elseif`. You will need to use OR (`||`), e.g.

```
if (month==1 || month==3 || month==5 ...)
```

(and then either 2 `elseif` bits, or 1 `elseif` and one `else` in the `if ... structure`).

---

<sup>5</sup> **MATLAB** does not allow the number of rows to differ, from column to column – a matrix must have a strictly rectangular shape. **MATLAB** and other programming languages allow the creation of *objects*, that are more flexible.

To TEST YOUR UNDERSTANDING ... for the matrix:

$$A = \begin{pmatrix} 4 & 66 & 13 & 42 & 36 & 14 \\ 33 & 4 & 0 & 28 & 11 & 22 \\ 18 & 26 & 7 & 1 & 5 & 19 \\ 12 & 9 & 23 & 30 & 7 & 2 \\ 0 & 0 & 2 & 0 & 15 & 33 \\ 14 & 42 & 17 & 27 & 8 & 0 \end{pmatrix}$$

determine ... NOT using `find` (or similar), but rather a nested loop, how many occurrences there are of values <sup>6</sup> <sup>7</sup>:

1. greater than 9
2. greater than 9 but less than 20

Make a new *script m-file* for this. You'll need to create a nested loop to test each and every location in the *array* in turn. Display (`disp`) the result at the end (after the (nested) loop has ended). You can either set fixed loop limits, e.g.

```
% row loop start
for row=1:6
    % column loop start
    for column=1:6
        %%% CODE
    end
end
end
```

or you can be more clever/flexible and use the `size` function to determine the number of rows and column, e.g.

```
[nrows,ncolumns] = size(A);
% row loop start
for row=1:nrows
    % column loop start
    for column=1:ncolumns
        %%% CODE
    end
end
end
```

which now becomes generic for any sized array *A*.

For e.g. the number of occurrences of values greater than 9, you will need a counter, which you initially set as zero, before the first loop starts ... and then you increment ... i.e. add 1 to its value and re-assign the new total back to itself ... when the condition (value > 9) is met:

```
if A(row,column) > 9
    count = count + 1;
end
```

(This conditional test of whether the value of the current array element is greater than 9 or not, will go within the innermost loop.)

<sup>6</sup> Hint: Before the next loop starts, you'll need to define a parameter to keep count of the number of values you find that meet the criteria, and set it to zero. Then in the (nested) loop, increment the counter variable by 1 each time you find a value matching the criteria.

<sup>7</sup> Also hint: At the start of the script (after your initial descriptive comments!), define *A*.

146 str = 'do you like bananas?'

NEXT: for the simple tic-tac-toe ( $3 \times 3$ ) grid, at each (*column,row*) location, you are going to draw a colored square.

Firstly, at the start of a new *script m-file*, add:

```
% *****  
% YOUR COMMENTS ON WHAT THE PROGRAM DOES  
% *****  
% create a new figure window  
figure;  
% create a set of invisible axes that will the window  
fh = axes('Position',[0 0 1 1],'Visible','off');  
% scale the axes (to go from zero to 3)  
axis([0 3 0 3]);  
% hold on!  
hold on;
```

Here:

- The line starting `fh = ...` creates a plotting area with no axes visible, and filling the Figure window area (`[0 0 1 1]` in normalized units). The handle to this is returned (variable `fh`), just in case we ever need it later.
- Then, the axes are scaled for convenience – there are 3 rows and 3 columns in the grid we want to create, so a 'reasonable' choice is to set `axis([0 3 0 3])`, although we need not have.
- You know what `hold on` does, right ... ?

You can then add the nested loop code framework<sup>8</sup>:

```
% loop 1 start  
for column=1:3  
    % loop 2 start  
    for row=1:3  
        %% CODE  
    end  
end
```

<sup>8</sup> To your *m-file*, after the `hold on` line.

Note that for convenience and to relate things to a more familiar (*x,y*) coordinate system, we are now going to assume that `column=1` corresponds to  $x=1$  in the plot, and `row=1` corresponds to  $y=1$  in the plot. i.e. we are working from the bottom left hand corner, first across the columns, and then up the rows. (It does not matter what array location in terms of (*rows,columns*) you assume any (*x,y*) location corresponds to, as long as you remember what correspondence you are assuming.)

To draw a square, the easiest function to use is `patch` (see earlier). For the coordinate parameters to be passed to `patch`, if your current location in the loop is `column=1, row=1` – now taking the notation and orientation where we start counting from the bottom left-hand corner – the coordinates for the square are:

```
(0,0), (1,0), (1,1), (0,1)
```

and for which patch will then take input:

```
patch([0 1 1 0], [0 0 1 1], 'black');
```

(remembering that patch takes a vector of all the  $x$ -coordinates as a 1st parameter, and then a vector of all the  $y$ -coordinates as the second parameter).

If you do no more than this (and use the patch line exactly as written above in your code), you end up looping through the  $(3 \times 3)$  grid, but only even (re-)plotting the same square in the bottom left-hand corner ...

The mental leap is to generalize the problem and to notice that if your column ( $x$ ) and row ( $y$ ) values correspond to the loop variables column and row, respectively, you could write:

```
patch([column-1 column column column-1], ...
      [row-1 row-1 row row], 'black');
```

Try this (adding this patch command to your code). You should end up with 9 black squares in a  $(3 \times 3)$  grid ... which will simply look like a huge black square filling the figure window, as per Figure 4.6 ... :o)

You could make it a little more interesting by creating a color value derived from the values of the column and row counters, e.g.

```
color = (column + row);
```

and then modify the patch command:

```
patch([column-1 column column column-1], ...
      [row-1 row-1 row row], color);
```

Re-running the script, you now get Figure 4.7.

You might notice that the colors are the same along diagonals, because you get the same value of color whether you are at location (1,2) or (2,1). We could make the location colors more distinct by modifying how we derive a value for color, e.g.

```
color = (column^2 + row);
```

(Figure 4.8)

You could also create distinct colors by using the rand function. e.g.

```
color=[rand rand rand];
```

where we are now specifying random values (in the range 0-1) for each of the three red, green, blue color intensity components (see earlier). (No figure example shown.)

Maybe play about a little creating different patterns of colors.

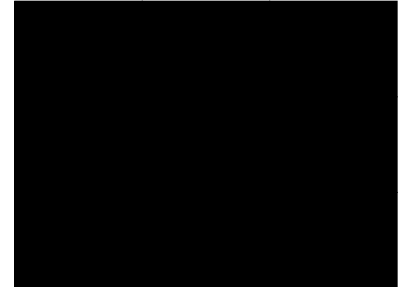


Figure 4.6: 3x3 grid of black squares ...

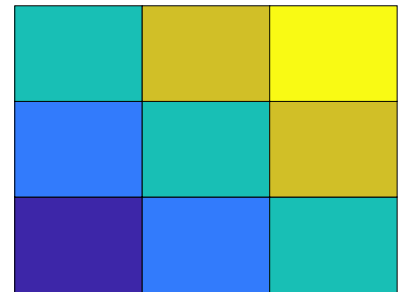


Figure 4.7: 3x3 grid of colored squares.

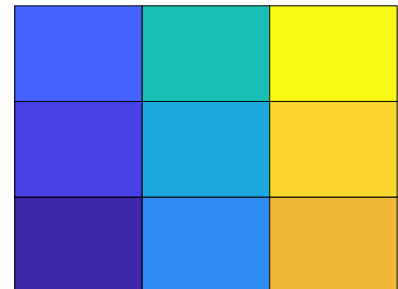


Figure 4.8: (yawn)

#### rand

rand (with no passed parameters), returns a quasi random real number, in the range 0.0 – 1.0.

This can be scaled, so e.g.  $10.0 * \text{rand}$  returns a number in the range 0.0 – 10.0.

$1.0 + 9.0 * \text{rand}$  returns a real number in the range 1.0 – 10.0.

$\text{round}(0.5 + 9.99999 * \text{rand})$  returns an integer in the range 1 – 10.

(Remember, that having obtained a random integer starting from 1, you can use this to index an array and hence ultimately, access different images at random.)

```
148 str = 'do you like bananas?'
```

AS A FINAL EXAMPLE, CONSIDER THE CHESS BOARD. A chess board consists of squares in a  $8 \times 8$  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
```

(taking  $m$  as the counter along the  $x$ -axis).

Now, if you wanted to define 8 squares up 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  $8 \times 8$  ( $n, m$ ) grid. Create a new *script m-file* and add this code to it.

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`.<sup>9</sup>) This array will be defined at the start of your program – after any comment lines and before the first loop starts.

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.<sup>10</sup>

To implement this in code – as we loop through both column ( $m$ ) and row ( $n$ ) on the board, we will 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`. i.e. if the number is divisible by 2 with no remainder, the number is even; if the remainder is 1, the number is odd.

#### `zeros`

`zeros` creates an array of dimension 2 or higher, consisting entirely of zeros! Actually, this is not as useless as it sounds, and represents a simple way to create a large array of a particular shape that can have then have (non zero) values set subsequently. To generate an  $n \times m$  matrix of zeros, you use:

```
A = zeros(n,m);
```

There is a short-cut if the 2 dimensions are the same (i.e.  $n = m$ ), and you can simply write:

```
A = zeros(n);
```

Simply list additional comma-separated integers (or variables containing values), to extend to 3 (or more) dimensions.

<sup>9</sup> 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*.

<sup>10</sup> Look up a picture of a chess board to convince yourself that this works.



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
```

Given that the mod 2 of an integer can only be 0 or 1, we could also write this:

```
board = zeros(8);
for m=1:8
    for n=1:8
        if ((mod(m,2)==1 && mod(n,2)==1) || ...
            (mod(m,2)==0 && mod(n,2)==0))
            board(n,m) = 1;
        end
    end
end
```

Spend a little time working through how this all works.

If it is easier to see, you could also expand the `if ...` into its individual parts, e.g.

```
board = zeros(8);
for m=1:8
    for n=1:8
        if ((mod(m,2)==1 && mod(n,2)==1))
            board(n,m) = 1;
        elseif (mod(m,2)==0 && mod(n,2)==0)
            board(n,m) = 1;
        else
            board(n,m) = 0;
        end
    end
end
```

Here, the final `else` bit is not strictly necessary as the `board` array has been initialized as being all zeros. (But it does not hurt to reassign a zero value if it helps you better read and understand the code.)

To visualize the result (array contents), after the nested loop has ended, you could use the e.g. `imagesc` function (cf. Figure 4.9). Beautiful.

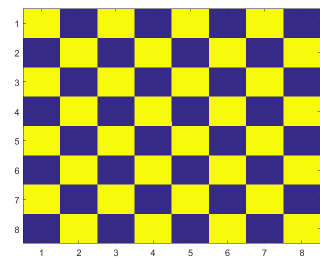


Figure 4.9: Chess board grid pattern.

```
150 str = 'do you like bananas?'
```

## 4.2 Algorithms and problem-solving

This (algorithms<sup>11</sup> and problem-solving) is not something that can really be 'taught' *per se*, but rather practised and aided by a logical state of mind. We'll go through a series of step-by-step examples. Hopefully this will also illustrate some general coding approaches.

### 4.2.1 Example #1: *max(!)*

So yes, this is a built-in *function* `max` in **MATLAB**, but suspend disbelief for a moment ... and pretend that there is not one. What if we wanted to create one, i.e. a *function* that is passed a vector of numbers, and returns the maximum value on that vector? <sup>12</sup>

So already, from the definition of the problem, you know how to create a new **m-file** a define a *function* – one that takes as input a variable (a vector), and returns the largest value in that vector.<sup>13,14</sup>

```
function [s_out] = maxxx(v_in)
% maxxx
%
% Takes a (single) vector as input;
% returns the maximum value.
%% CODE
end
```

You have hence created a (empty) shell for the program (*function*). You could try calling it/running it at the command line, just to check there is no problem so far, although it is clearly not going to do anything. To call your *function* at the command line – remember that you have defined the *function* to take 1 input – you must therefore give it an input ... First – make a test input in the **MATLAB** workspace (not in the *function*), e.g.

```
» A = [1:100];
```

which defines a vector of integers between 1 and 100. You can now call the *function* at the command line, passing in this test vector to `maxxx`:

```
» maxxx(A)
```

Nothing gets returned yet, but equally, you should not be presented with pages of red **MATLAB** error text.

You could extend your basic testing by returning just the first element of the input vector, i.e., before `end`, add:

```
s_out = v_in(1);
```

and then re-try:

```
» maxxx(A)
```

<sup>11</sup> *algorithm* – "a process or set of rules to be followed in calculations or other problem-solving operations"

<sup>12</sup> **Example codes provided**

<sup>13</sup> Note that you do not need to specify that the input variable is a vector, just that there is a variable input.

<sup>14</sup> Here, my personal naming convention has:

`s_out` – the output variable, and `s` for scalar

`v_in` – an input variable, with `v` designating vector ...

(This is perhaps, overkill, but leads little room for any confusion later.)

You are still not solving the problem yet, but in this step, you have now demonstrated that your *function* can take in a (*vector*) input, do something with it, and now set the output variable (and return a scalar).

OK, so ... we need to devise an algorithm to find the largest value in the vector `v_in`. The first piece of potentially useful information you can find, is the number of elements in the vector. You can obtain this via the function `length`<sup>15</sup>.

So near the top of the *function* (but below the function definition line and the following comment lines), you could create a *variable*, set equal to the number of elements in the *vector* that you are going to have to process:

```
nmax = length(v_in);
```

What about the next part or structure in the program? You are going to need to search through all the elements of the vector if you are going to find the maximum value, so presumably a *loop* is required – one that loops through from the first to last element of the vector:

```
for n = 1:nmax,
end
```

(and which goes after the value of `nmax` has been defined).

The crux of the problem is recognising that you need to keep tabs on the running maximum value, or a local or temporary maximum value, that is your maximum value so far, as you progress through repeated iterations of the *loop*. With this value, you are going to test whether each element of the array is larger than it – if true (the element in the vector being tested is larger than the current or largest-to-date maximum estimate) – you are going to replace the current maximum estimate with the vector element that you have just found is larger than your largest so far. We could call this *variable*, e.g. `temp_max` to indicate that it is temporary and not necessarily the largest value of the vector as a whole.

Within the loop, the test we make is therefore:

```
if (v_in(n) > temp_max),
    temp_max = v_in(n);
end
```

Almost there. Run the program/*function* and see what happens.

**MATLAB** is unhappy about the line where the value of `temp_max` is being tested against `v_in(n)`. It may be obvious to you when (in terms of loop iterations) this is occurring. If not, why not, just after:

```
for n = 1:nmax,
```

Which in this example just happens to be the first element of the vector

<sup>15</sup> It turns out that `length` does not care about the orientation of a vector, and:

```
A=[1:10];
length(A)
```

gives the same answer (10) as:

```
A=[1:10]';
length(A)
```

152 str = 'do you like bananas?'

disp(lay) the value of n. OR, add a breakpoint on the problem line, so that **MATLAB** will pause just before the line that gave the error, is executed.

Either way, you should have found that the value of n is 1, i.e. the error is occurring on the very first iteration of the loop. Why? As per the error – **MATLAB** does not know what temp\_max is. This occurs because you have not yet assigned it a value when the loop starts.

So our problem is one of initialization – we need to give temp\_max an initial value, so that when the first iteration of the loop occurs, and the first element in the vector is accessed, there is something to compare it with.

There are 2 (probably 9999999) ways to go about this:

1. Seed the value of temp\_max with a value so improbably small, that you are betting that any conceivable array of numbers with have a number greater than this.<sup>16</sup> For example, before the loop starts, you might write:

```
temp_max = -999999999;
```

And then go through testing all nmax elements in the vector.

2. Better, would be to initialize your temporary maximum variable with the first element in the vector, i.e..

```
nmax = v_in(1);
```

You might also recognise that you need not test this against the first element, and the loop could now start at 2:

```
for n = 2:nmax,
```

Finally, remember to set the output variable equal to the maximum value that you find.<sup>17</sup>

There are 2 further testing or debugging (if you have issues) steps:

1. Firstly, simply go through in your mind, what you think happens on each iteration. Writing down how the values, e.g. of the temporary variable change on each iteration, is a good idea. Obviously you can do this prior to writing the code, to give you an idea of how it will work (or not).
2. Create a series of test arrays (vectors), or varying length, ordered, or random numbers, integers and/or reals, whatever youn like ... and see if your *function* works each and every time.

---

WHEN YOU ARE PASSABLY HAPPY WITH THAT – write a (new) *function* that finds the *minimum* element in a vector input.<sup>18</sup> Call this function minxxx. **You will need this later on ...**

---

<sup>16</sup> There are obvious dangers here, should a vector of all insanely low values be given as an input. You could for instance determine whether any of the values were higher than the seed value, and if not, report or return an error message. So in this case, even if the function did not work, you would be told why. A bit like **MATLAB** functions in general, no?

<sup>17</sup> After the loop ends.

<sup>18</sup> Easiest is to copy the file, rename it minxxx.m, and edit the *function* name on the first line of the file. Then ... edit the code to find the minimum rather than maximum value.

NEXT ... AS A 3RD FUNCTION – create a new *function* that now returns a second variable from the *function* – equal to the number of elements that are equal to the maximum.<sup>19</sup> (i.e. if the maximum value appears 5 times in the input vector ... you additionally return the number '5'.)

The structure of the (your new) *function* will now look like:

```
function [s_out1 s_out2] = maxxex(v_in)
% maxxex
%
% Takes a (single) vector as input,
% returns the maximum value
% PLUS the number of elements equal to that value.
%%% CODE
end
```

which passes back two variables (rather than the single one you had before).

You might try exploring what you are doing to 'do' with 2 outputs ... e.g. if you (temporarily) added the following lines to your *function*:

```
s_out1=1.1;
s_out2=2.2;
```

then at the command line, you could test the function:

```
» maxxex([])
```

here passing in an empty vector (as we are not yet doing anything with the input), and you should see:

```
ans =

    1.1
```

which ... is only 1 of the outputs!!! So you need to explicitly assign the result of your function to 2 variables, e.g.

```
» [a b] = maxxex([]);
```

Now you get the results you were expecting (1.1 and 2.2) returned and assigned to the variables *a* and *b*, respectively. Note that as you are assigning the result of your function to variables, you can add the *;* to the end of the line to suppress unnecessary output at the command line.

Back to the construction of the algorithm and its coding ...

The code to generate the first variable (*s\_out1*) – the maximum value of the elements in the input vector – is as before. You only need to work out the code to determine the value of *s\_out2*. The key to this is recognising that previously, when the value of *temp\_max* was equal to *v\_in(n)*, you did nothing, as you were only

<sup>19</sup> If we were cheating, and we are not ... then one could use the built-in **MATLAB** functions as so:

```
length(find(A==max(A)));
```

Which says ... find the elements in *A*, equal to *max(A)* and determine the *length* (number of elements) of this resulting vector (*find(A==max(A))*) ...

interested in `v_in(n) > temp_max` and hence updating (replacing) the current value of `temp_max`. What you need is an `elseif` statement, and test whether you have found a second value in the vector equal to `temp_max`. If so, you need to somehow record this occurrence.

A partial solution (i.e. a first next step in the development of the code) might look like:

```
% Loop through all
% but the first element in the vector
for n = 2:nmax,
    if (v_in(n) > temp_max),
        temp_max = v_in(n);
        temp_n = 1;
    elseif (v_in(n) == temp_max),
        temp_n = 2;
    end
end
```

where `temp_n` is where we keep a track of the number of maximum elements – setting this to a value of 1, when we first find a new largest value in the vector and then a value of 2 if we find a second occurrence of that value.

But this does not quite work. Why? Try throwing some test vectors at it, e.g.

```
A = [1 5 7 3 8 2 4];
B = [1 5 7 3 8 2 4 3 5 8];
C = [1 5 7 3 8 2 4 3 5 8 7 7 8];
```

In particular – is the answer to C ... right?

So far, we have accounted for a duplicate (maximum) value, but the solution (and code algorithm) is not *general*, i.e. we do not handle the general case of there being  $n$  duplicate maximum values in the vector. A more general solution looks like:

```
% Initialize duplicate counter
temp_n = 1;
% Loop through all
% but the first element in the vector
for n = 2:nmax,
    if (v_in(n) > temp_max),
        temp_max = v_in(n);
        temp_n = 1;
    elseif (v_in(n) == temp_max),
        temp_n = n+1;
    end
end
```

Here we have to seed/initialize the value of `temp_n` because we start by setting `temp_max` as equal to `v_in(1)`, i.e. we already have 1

instance of the value of `v_in(1)` being the maximum, by the time the loop starts.

Again – a key to programming and developing algorithms, is to follow the behaviour of the code in your head (as well as adding break points and testing with a variety of inputs, including extreme assumptions). For instance – assume that the 2nd element in the vector was equal to the first, and follow the code around – check that the value of `temp_n` is incremented appropriately and the output, if the vector was only 2 long, or there were no larger values in the remainder of the vector, is right. What if the 1st 3 elements were all equal? How does that pan out in terms of behaviour and output?

In general – if the code works for a selection of extreme assumptions, it will generally work. Use a combination of your head (and paper and pencil) and testing (and debugging if necessary).

#### 4.2.2 Example #2: *sort(!)*

(Yes, **MATLAB** also has functions for sorting values in an array ...)

IN THIS SECOND SET OF EXAMPLES – imagine that you have a vector of numbers, and you wish to sort them into ascending order. The *function* would take a vector as input, and return a vector of the same length as output, comprising all the values of the input vector, but now sorted in order.

How to go about this?

Well – first create the function framework, as before:

```
function [v_out] = sortx(v_in)
% sortx
%
% Takes a (single) vector as input,
% returns a vector of the same length, with
% all the values sorted in ascending order.
%%% CODE
end
```

So far ... so good. What do we need to do within the *function*?

Well, we should start with a simple text of the *function* structure. For instance, we could start by finding the minimum value in the array, and placing it at the start of a new array – the one that will form the output. In this test, in place of `%%% CODE` in the empty *function* outlined above would go:

```
% Initialize the output vector (as empty)
v_out = [];
% Find minimum value
x = minxx(v_in);
% Update output vector
v_out = [v_out x];
```

```
156 str = 'do you like bananas?'
```

remembering that one way to build up a vector for output, is to append to (concatenate) a value to an existing vector. The vector must start defined as something ... here, as empty `[]`.<sup>20</sup>

Try this out at the command line to check that it does indeed return the minimum value of the vector input:

```
» sortx([1 2 3 4 5 6 7 8 9])
```

Of course, you could create a more complicated and challenging vector to test it with and rather than pass the values directly, e.g. if your vector of numbers to sort was assigned to variable `testvec`, then you would type:

```
» sortx(testvec)
```

So far, your function returns the lowest (1st) value, rather than sorting them all. You might spot that the 3rd through 6th lines of code, need to be repeated – i.e. you keep calling the find-the-minimum function, and adding this value to the output vector. But if this is all you did, the program would run forever and return an infinite number of repeats of the minimum value (or give you a vector the length of the input vector with identical (minimum) values in, depending on how you set up the loop (e.g. `for` or `while`)) – this is because once you have found the minimum value, and added it to the output array, the value still exists in the input array and will be found again (and again and again) if you create a loop around it.

The key ... at least, the key to creating the particular algorithm ... is to remove the element that you have just used from the input array. e.g. if we find a '1' and this is the lowest value, after it has been added to the output array, it needs to be deleted from the input array. This means that we need to know 'where' in the (input) array the minimum value was.

So ... your next task is to modify your `minxx` function, to return the position in the vector that the minimum value occurred at (in addition to the value of the minimum value itself).<sup>21</sup> i.e. your `minxx` (or whatever you called it) function needs to look like:

```
function [s_out1 s_out2] = minxx(v_in)
```

where `s_out1` is the minimum found value, as before, and `s_out2` is the index position of that number in the vector.

Having obtained the index of the minimum value – assuming that you passed it into a variable `m` (and with `x`, as before, the minimum value), e.g.

```
[x m] = minxx(v_in);
```

you can now delete this entry from the input array by means of:

<sup>20</sup> Also in this code fragment – you need to have created that minimum finding function ...

<sup>21</sup> HINT: When you find a new minimum value in the loop, the index (the position of that minimum value) is `n`. So you need to create a temporary variable that you update with the value of `n`, only when a new minimum value is found in the loop. You will see this variable with '1', consistent with assigning the first minimum value from index 1 in the vector.



```
v_in(m) = [];
```

where  $m$  is the index.

The code within your modified sort (not `minxx`) *function*, would now look something like<sup>22</sup>:

```
function [v_out] = sortx(v_in)
% sortx
%
% Takes a (single) vector as input,
% returns a vector of the same length, with
% all the values sorted in ascending order.
% Determine number of elements in vector
nmax = length(v_in);
% Initialize the output vector (as empty)
v_out = [];
% Loop through all but the first element in the
vector
for n = 1:nmax,
    % Find minimum value
    [x m] = minxx(v_in);
    % Update output vector
    v_out = [v_out x];
    % Remove used element
    v_in(m) = [];
end
% function end
end
```

<sup>22</sup> Note the notation for catching the 2 returned values from the `minxx` function:

```
[x m] = minxx(v_in);
```

---

FOR (EASY) FUN: create a (new) *function* that sorts in descending order.

---

FOR EVEN MORE (BUT LESS EASY) FUN: create a (new) *function* that sorts in descending order ... but ... excludes duplicate values, i.e. no 2 values should be the same in the output vector.<sup>23</sup>

<sup>23</sup> HINT: modify the `maxxx` function to return the index of all the elements having a value equal to the maximum value. This will be a vector, which will be appended to as the loop progresses, with each value being assigned the current value of  $n$  (the position of the maximum value in the vector).

### 4.2.3 Example #3: a gridded problem

We are going to base this next example around the (modern) topography of a simple Earth system model (**GENIE**).

Load in the file: `model_grid.txt`<sup>24</sup> in the 'usual way'. Briefly check out the new array in the Variable window. If you were told that values 1 through 16 represented ocean cells<sup>25</sup>, and values above 90, land<sup>26</sup> – it is possible to make out the shape of the continents visually in the pattern of numbers in the array (albeit they are rendered at low spatial resolution). The grid of numbers can also be visualized using the `image` function (see earlier). See if you can specify the scaling in such a way that you can render the ocean topography reasonably well, e.g. as per Figure 4.10.

What you are going to do is to draw this grid ... using the `patch` function. We'll simplify things and assume that each cell is 1 unit wide and 1 unit high – i.e. the grid goes from 0-36 units in both longitude (x-axis) and latitude (y-axis) directions. In fact – lets forget entirely about longitude and latitude for now.

Ultimately, the point of this exercise is to draw land as grey cells, and assign the ocean cells a color according to their depth. But lets start by drawing a grid of cells (of any color).

So how to start? Make yourself a new *script* (.m) file. I guess open a figure window, set `hold on`, and set the axes from 0-36 in both directions.

To begin with, simply draw a single cell (the first cell of what will become the grid). It should appear at the bottom left corner of the plotting area. If you find you have an odd shape appearing ... you have got a set of 4 y-coordinate values that is inconsistent (/out of sequence) with the y-coordinate values. Draw out the coordinates on a piece of paper and/or write down the 4 vertices of the rectangle, to help visualize.

You know that you will need to draw a row of 36,  $1 \times 1$  squares using `patch`. So, make a loop ...

```
for i = 1:36,
    ...
end
```

(here: *i* for longitude). And then draw a series of squares, each with their right-hand edge corresponding to the value of *i* (so the left-hand edge is at (*i*-1)). For now, draw only a single row of cells, with the y/latitude-axis (which I will use index *j* for) from 0 (bottom edge) to 1 (upper edge).<sup>27</sup>

The key step here is to add in *i* into the list of x-coordinates, so that the x values progressively increase as the loop proceeds. (Leave the y-coordinate values alone for now.) Again – if you have odd

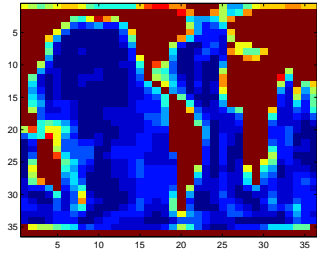


Figure 4.10: Ocean topography (blues through red) in the 'GENIE' Earth system model. Land is shown marked in brown.

<sup>24</sup> From week #1.

<sup>25</sup> If you must know (but you don't need to know it at all): the lower the value, the deeper that part of the ocean, with 1 representing the very deepest ocean floor, and 16 the shallowest.

<sup>26</sup> The values: 91, 92, 93, 94, represent different compass directions of runoff on land. (another not interesting and barely useful fact.)

<sup>27</sup> To make the individual cells more apparent, you can specify a different edge color, and also make the edges thicker, e.g.

```
patch([0 0 1 1],[1 0 0
1],...
'k', ...
'EdgeColor','red', ...
'FaceColor','yellow', ..
'LineWidth',2);
```

(why it needs the 3rd, color option set, when the face and edge are set separately ... is one of life's little mysteries ...)

shapes appearing ... you have got an inconsistent sequence of coordinate values in the x and/or y vectors. Try substituting `i=1` into your list of 4 x-coordinate value and see whether you get the expected list of 4 (x,y) pairs. Then try substituting `i=1` in. (This can all be done on paper if you like.)

Now, you need to draw the other 35 rows of cells above this.

Think about this for a moment – you have a loop, drawing each of 36 square cells in a row ... in the first (bottom) row. Now you need to repeat the drawing of 36 rows ... a total of 36 times. This is a *nested loop*. Its form is:

```
for j = 1:36,
    for i = 1:36,
        ...
    end
end
```

meaning that whatever code goes in the very middle, is carried out  $36 \times 36$  times (and in fact there are  $36 \times 36$  cells in total to draw).

If your code to draw one line of cells was e.g.:

```
for j = 1:36,
    for i = 1:36,
        patch([i-1 i-1 i i],[1 0 0 1], 'r');
    end
end
```

you now need to modify the y-axis values in the `patch` command, so that they reflect the increase in the value of `j` as you move up to a new row (and the `j` loop progresses towards a value of 36).

See if you can get this working (just a  $36 \times 36$  grid of cells ... colors of your choice ...).

Once you have this working: get a coffee.<sup>28</sup>

Actually, you won't need a coffee<sup>29</sup> – this is the hardest part done.

First, for the land cell designation. In the inner loop, rather than just draw a colored square regardless of anything, you need to decide whether to draw a grey ([0.5 0.5 0.5]) or e.g. red square, depending on whether the model grid at that particular (*i,j*) location is land or ocean. Land is designated by a value above 90. So you need to test the value of `model_grid`<sup>30</sup> – greater than 90 results in a grey square being drawn, and less than or equal to 90 (or just `else`), a e.g. red square. Try this, and if the grid comes out upside down, or back-to-front or something, you know how to transform the array you have read in.<sup>31</sup>

Second, to assign colors that depend on the depth. Rather than a e.g. red cell, when the cell is ocean (90 or less), create a RGB color that is a function of the depth value.<sup>32</sup> This is where the concepts of algorithms comes in – they need not be long, complicated codes, but

<sup>28</sup> You will need it.

<sup>29</sup> The margin note above was an alternative fact.

<sup>30</sup> Or whatever you called the array when you loaded the data in.

<sup>31</sup> You might note that while we tend to think about plotting of lon-lat as (*i,j*), in **MATLAB**, *i* corresponds to rows (lat) rather than columns (lon). So it is helpful to flip the rows and columns of the array around, so we can write (*i,j*) as (lon,lat) (i.e. (x,y)). You might also find it is necessary to flip the array if it comes out up-side-down.

<sup>32</sup> In this particular model – depth goes from a value of 1 (deepest) to 16 (shallowest).

```
160 str = 'do you like bananas?'
```

can be simple equations that achieve the desired result. For instance, given the nature of the RGB scale, and that we have a scale of values from 1 to 16, what immediately comes to mind is:

```
vcol = [1 1 1]/model_grid(i,j);  
patch([i-1 i-1 i i],[1 0 0 1],vcol);
```

which has the effect of creating a grey-scale for the depth values from 1 (lightest) to 16 (darkest). The other way around would be:

```
vcol = 1.0 - [1 1 1]/model_grid(i,j);  
patch([i-1 i-1 i i],[1 0 0 1],vcol);
```

This particular algorithm for converting depth to a unique color, will clash with the grey coloring of the continents (which were assigned a mid-grey) for a certain depth. So try and devise a color scale (in color!).

---

#### HOW ABOUT MARKING ON THE CONTINENTAL OUTLINE?

1. The first task is to draw the grid – as per above.<sup>33</sup>
2. Then, you want a second nested (i,j) loop, within which you will test for a boundary between land and sea, and draw a line to delineate this segment of coastline.

There are a variety of ways to go about all this, some long with lots of duplicated code, and some cunning<sup>34</sup> and compact. We'll go for the ultra-crude approach, but leave it as an exercise for you to think about how it could be simplified and rationalized later on.

We'll take the case of the ocean being on the right hand side of a land (continental) cell (i.e. a East coast). We'll need to search through the entire grid and hence need a double/nested loop as before:

```
for j = 1:36,  
    for i = 1:36,  
        ...  
    end  
end
```

The plan will be<sup>35</sup>:

1. Test for whether the cell is land (value > 90).
2. If the above is true, test for whether the cell immediately to the right, is ocean.
3. If the above is (also) true, then we have found a border between land and ocean and just need to draw the border.

You have done the testing of grid point (cell) values before, in coloring land one color and ocean (depth) another:

<sup>33</sup> You want to draw the complete grid first, because if you draw on the coastline lines as you go, you may find that you end up partially obscuring a coast line with the next filled cell.

<sup>34</sup> So cunning in fact, that you could put a tail on it and call it a fox.

<sup>35</sup> Inevitably – you need to formulate a plan – your algorithm, first, whether simply in your head, or on paper.

```

if (model_grid(i,j) > 90)
    ...
end

```

To then test the grid point to the right:

```

if (model_grid(i,j) > 90)
    if (model_grid(i+1,j) <= 90)
        ...
    end
end

```

Here – `model_grid(i+1, j)` is the cell to the immediately to the right (greater longitude) than `model_grid(i, j)`.

It is then just a matter of identifying the start and end of the line that you will draw.<sup>36</sup>

The only one thing to note here, and if you have coded the loop as show above, you'll end up with an array out of bounds error reported by **MATLAB**. Think through what happens in the loop when the value of `i` reaches 36 – `i+1` is then 37, yet the array is only  $36 \times 36$ . So in the case of finding the East coast segments of the continental outline, you need to:

1. Only loop from `i = 1:35`, so that the value of `i+1` is always a valid array index.
2. Because you still need to determine whether at the edge of the grid, there is an East coast line, carry out a specific test for the edge of the grid:

```

if (model_grid(36,j) > 90)
    if (model_grid(1,j) <= 90)
        ...
    end
end

```

Here – if the cell at the far right edge of the grid (`i=36`) is land, we test whether the cell at the far left of the grid (`i=1`) is ocean.<sup>37</sup>

Note that this code fragment, because the value if `j` changes, goes within the outer, `1:36 j`-loop (but not within the `1:35 i`-loop).

The complete code for this search ... except for the actual drawing of the edge line, is:

```

for j = 1:36,
    for i = 1:35,
        % Search i from 1 to 35
        if (model_grid(i,j) > 90)
            if (model_grid(i+1,j) <= 90)
                % DRAW EDGE
            end
        end
    end
end

```

<sup>36</sup>Setting a thicker-than-default line width, e.g. `'LineWidth', 2` will help the continental outline stand out.

<sup>37</sup>Remember that the grid wraps-around in longitude.

```
162 str = 'do you like bananas?'
```

```
    end
end
% Special case of i=36
if (model_grid(36,j) > 90)
    if (model_grid(1,j) <= 90)
        % DRAW EDGE
    end
end
end
end
```

It remains for you to create a similar code for finding (and drawing) the West coast segments. And then, the North and South coast segments. Remember in this latter search – the grid does not ‘wrap-around’ and  $j$  need only from  $1:35$  and  $2:36$  (with no ‘special case’).

---

IN A FINAL, OPTIONAL, EXAMPLE ... of the bathymetry data – questions such as: ‘How many land cells are there? What fraction of the Earths surface is land?’, ‘What (area) fraction of land is within 70 m of the current sealevel?’, can be answered with 1, or at most, a few lines of code (and maybe a function call for the calculation of the area of a  $1^\circ$  grid cell). A more involved question might be: how many distinct land masses are there? Or: can we assign a label to them (assuming we want to in the first place).

Jumping straight into the full resolution 1 degree resolution dataset is probably not such a good idea, so instead, to start with, you are going to use the **GENIE** model grid/topography again. Further-more, you are only going to be concerned with the land-sea mask and not even worry about height above, or below, sea-level.

You are going to count up (and sequentially number) the different land masses<sup>38</sup>. Obviously, you could do this by eye for this particular example (but how about counting the unique land masses in the 1 degree topography dataset?). Think about how you are mentally ‘doing’ this – i.e. what processes are going through your brain (other than how long until the end of class) as you decide what makes any particular land mass distinct from another one. This may well inform how you go about coding and creating an algorithm to solve this.

A sensible start might be to loop through all the points in the grid. As you should have gathered – this can be done as a nested loop. To make it a littler cleverer: rather than setting in stone a specific count limit in the loops, which in this example would be 36 (for both longitude and latitude), you can extract the size of the array and hence the limits to the 2 dimensions by:

```
[n_lat n_lon] = size(model_grid);
```

<sup>38</sup> By ‘different’ – assume that distinct land masses (which here may be continents or just islands) are groups (or single) of land cells that share no common edges (excluding diagonal connections). The isolated block of cells representing Australia ia an obvious example.

Here: `size` returns the number of rows and columns of the array, corresponding to the number of latitude, and longitude bands, respectively. Your code (which should be placed in an `m-file`) will the start to look like:

```
topo = load('model_grid.txt','-ascii');
[n_lat n_lon] = size(topo);
lon=n_lon
  for lat=n_lat
    end
  end
```

but with ... suitable comments added of course ... By all means add some suitable debug lines and test it (the loop behaviour).

You are going to need an array, the same size as the topography dataset, to store the number assigned to each land mass, i.e. each grid cell needs to be labelled with a land mass number, and something distinct from this if it is not land at all (i.e. ocean). You can create an array of zeros easily with the **MATLAB** `zeros` function (see Box). Then as you raster through the grid (via the nested loop), you can assign land points a value corresponding to the land mass number, and leave the ocean points as zeros.

To get your hand in – first add to the code above, the creation of the array of zeros (this is going to need to come after you have determined the size of the data array and hence the values of `n_lat` and `n_lon`, but before the loop starts). Then, within the loop, test for whether or not the grid point is land or ocean (see above for what the values in the **GENIE** model topography array mean), and if the point is land, set the value to 1. Plot the results with `imagesc` and check that you get just 2 colors – one for ocean (0) and one for land (1). In fact, you could keep all this code and resulting array. Then for the array storing the land mass number, create a second array of zeros. (Remember to name the arrays something meaningful, not just `A`, `B`, . . . , and comment the code adequately.)

So how are you going to go about identifying new land masses and numbering them? You have to start somewhere, that somewhere will be designated by a 1 (the first land mass). How do you know that this is the first land point, and not the second? You could count up, for instance – each time you find a land point, you *increment* a counting variable by one, e.g.

```
n_runningtotal = n_runningtotal + 1
```

remembering that at the start of the code, you need to initialize the value of `n_runningtotal` to zero.

This is not quite what you want, for instance, if you run the following:

```

topo = load('model_grid.txt','-ascii');
[n_lat n_lon] = size(topo);
land = zeros(n_lat,n_lon);
land_id = zeros(n_lat,n_lon);
n_runningtotal = 0;
for lat=1:n_lat
    lon=1:n_lon
        if (topo(lat,lon) > 90)
            n_runningtotal = n_runningtotal + 1;
            land_id(lat,lon) = n_runningtotal;
        end
    end
end
end

```

you should get all the land points numbered in turn (check this), but not with land points grouped into continuous regions with different numbers assigned only the distinct land masses. So ... it is getting closer, but it is still missing something.<sup>39</sup> (It is quite pretty to plot though, as per Figure 4.11. Perhaps also try the 2 loops the other way around, with the `lon` loop first and outermost, and see what happens (/is different about it).)

As you might imagine, the crux of the algorithm is how to assign a new identifying land mass number to a land grid point only when it does not connect to a land point which already has a number – in this case, the same value for the identifying number needs to be used. In other words: if a newly found land point connects to a land point with the identifying value 5, then the new point also needs to be labelled with a 5. So ... and here is the critical bit ... we need to 'look around' each new grid point to see if there is an already labelled point immediately next to it. Pause and think about this. Maybe mentally, or on paper, work your way through the start of the grid, label the first land point you find, and work out what the mental steps are upon finding the next land point, to see if it needs to be assigned a new number, or not (and is instead connected to a point which already has a number). This mental/conceptual step is important and hopefully will lead you to a suitable and working *algorithm* that can be written down in code. In essence, all you are going to be doing is encoding (in code), using conditional tests and perhaps further loops, the mental steps that you are going through<sup>40</sup>.

OK. So how exactly are we going to go about it? There is a really clever way, but we'll skip over that :o) And, a crude and simple way, but one that will still solve the problem (although it will turn out that we will require additional steps – one to get most of the way there and then several to make minor corrections to the initial algorithm). We are going to keep the counting variable, but now only update it (increment it by one) if we need a new land mass number.

<sup>39</sup> This is not a bad way of working in fact – get something of a likely correct form (e.g. nested loop in this case, setting up some arrays of zeros, creating a counter) but not quite getting the answer going first, then refine to get it doing what you actually want.

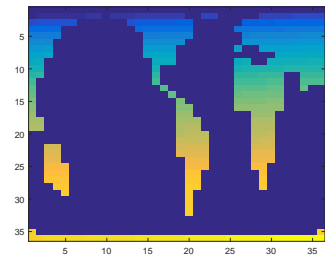


Figure 4.11: The 'GENIE' mode land grid, with land points assigned a sequential integer (working across and down the grid – from West to East, and then North to South).

<sup>40</sup> Unless you are just thinking about `icecream`.

```

icecream
There is no icecream function in
MATLAB. I checked. In fact, rather
sadly, MATLAB tell me:
    icecream not found.

```



So, \*in practice\* then, how are we going to decide if the counter is incremented and hence what value to assigned to a particular cell?

First, we need to test whether the current cell is ocean or land:

1. If ocean – do nothing, and leave corresponding value in the land mass array at zero.
2. Else (if land) – we need to work out what value to assign to the cell in the land mass array, by:
  - (a) If an adjoining cell is land and has been assigned a value in the land mass array, then assign the same value to the current cell.
  - (b) If all adjoining cells have a zero value, either because they are ocean, or because they have not been assigned a (non-zero) value yet (because the loop has not yet reached that far in the array), then increment the counter and assigned the cell this new number.

This simple decision tree is something that you could draw a flow-chart for if it helps. Also work through in your mind to see if it appears to 'work'.

The next step is coding the 'look around' (the current grid cell) bit. Actually, if you think about it, you need not look at the adjoining cells in all of the N, S, E, and W directions, because if we are looping through the grid such that we raster across the grid from left (W) to right (E), and then from the top (N) to bottom (S), cells to the E and S of the current grid point have not been reached yet and so must have a zero value. Hence you only need to interrogate the value of cells to the W and N of the current position (as defined by `(lat, lon)`). You can write the conditional test for the adjoining cells being zero (and hence ocean, as they must have already been visited and hence left with a zero value), by<sup>41</sup>:

```
if ( (land_id(lat-1,lon)==0) && (land_id(lat),lon-1)==0)
)
end
```

It should be obvious that this is testing for the cell immediately to the North (`lat-1`) \*and\* the cell to the West (`lon-1`), both being zero.

Naturally, your first attempt does not work! Why? Think through what happens as you start to make your way through the grid. You only have to think through what happens at the very first grid point in fact. The first grid point is `(1, 1)` yet you are testing cells with indices of `lat-1` and `lon-1` ... which will be zero and hence not a valid array index<sup>42</sup>. So you need to avoid testing for `lat-1` if `lat==1`, and avoid `lon-1` if `lon==1`. There are a variety of ways

<sup>41</sup> Not all of these parentheses are necessary – I have written it like this to make the conditional (hopefully!) completely clear.

<sup>42</sup> **MATLAB** array indices always start at one. (Whereas in **FORTRAN**, it is possible to start counting the array rows or columns from zero, or even a negative number.)

```
166 str = 'do you like bananas?'
```

of structuring this, some using more and some less, code. One possibility (and not necessarily the most optimal one) is:

```
if ( (lat==1) && (lon==1) )
    % on both Western and Northern edges (top LH grid
    corner)
    CODE BLOCK #1
elseif (lat==1)
    % on Northern edge
    CODE BLOCK #2
elseif (lon==1)
    % on Western edge
    CODE BLOCK #3
else
    % cell lies neither on Western nor Northern edge
    CODE BLOCK #4
end
```

In 'CODE BLOCK #1', you will simply need to increment the land mass counter and assign the cell this value<sup>43</sup>. 'CODE BLOCK #4' will use the conditional code that you saw earlier:

```
if ( (land_id(lat-1,lon)==0) && (land_id(lat),lon-1)==0)
)
end
```

<sup>43</sup> This will be executed only once (assuming that the cell is land) because there is only one situation in which both `lat` and `lon` can have a value of one – the top LH corner of the grid.

and when this is true, increment the land mass counter and assigned the cell this value. But as part of this conditional structure, you will also need to test the values of the cells to the North and the West individually. If either has a non-zero value, assigned this value to the current cell (and do not increment the counter).

The remaining 2 pieces of code are sort of half way between #1 and #4, and will be conditionals testing for the situations:

```
land_id(lat-1,lon)==0
```

(#2) and having already excluded the possibility of both `lon` and `lat` being equal to one, or:

```
land_id(lat,lon-1)==0
```

(#3) (having excluded the possibilities that firstly that `lon` and `lat` are both equal to one, but also that `lat` is equal to one (and implicitly; `lon` is greater than one)). In both cases you only need to test the value of one adjacent cell (and if zero, increment the counter etc., or use the adjacent cells value, otherwise).

The code is inherently simple, but there is now lots of it and a big chunk of code with lots of conditionals can look intimidating and difficult to debug or understand. The key is to work through it with a couple of example (`lat,lon`) loop values and test what it does under these conditions, verifying that the algorithm is doing what it should.

The complete code that tests the value of the surrounding cells and on the basis of this result, assigns a land mass value, looks like:

```

if ( (lat==1) && (lon==1) )
    % on both Western and Northern edges (top LH grid
corner)
    n_runningtotal = n_runningtotal + 1;
    land_id(lat,lon) = n_runningtotal;
elseif (lat==1)
    % on Northern edge
    if ( land_id(lat,lon-1)==0 )
        n_runningtotal = n_runningtotal + 1;
        land_id(lat,lon) = n_runningtotal;
    else
        land_id(lat,lon) = land_id(lat,lon-1);
    end
elseif (lon==1)
    % on Western edge
    if ( land_id(lat-1,lon)==0 )
        n_runningtotal = n_runningtotal + 1;
        land_id(lat,lon) = n_runningtotal;
    else
        land_id(lat,lon) = land_id(lat-1,lon);
    end
else
    % cell lies neither on Western nor Northern edge
    if ( land_id(lat,lon-1)~=0 )
        land_id(lat,lon) = land_id(lat,lon-1);
    elseif ( land_id(lat-1,lon)~=0 )
        land_id(lat,lon) = land_id(lat-1,lon);
    else
        n_runningtotal = n_runningtotal + 1;
        land_id(lat,lon) = n_runningtotal;
    end
end
end

```

and sits within the double loop and test for a land cell:

```

for lat=1:n_lat
    lon=1:n_lon
        if (topo(lat,lon) > 90)
            CODE
        end
    end
end
end

```

Really, it is not as bad as it looks! Much of the code is simply dealing with the special cases of the grid point being on one or other or both, of the W/N grid boundaries. Without this, the generic code for the rest of the grid is simple (the block labelled % cell lies neither on Eastern nor Northern edge).

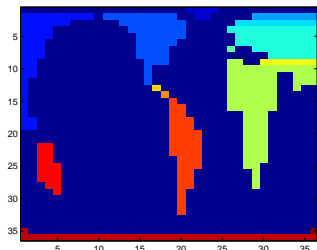


Figure 4.12: The 'GENIE' mode land grid, with land points assigned a unique identifier ... almost ... (!)

If you complete the code with the file loading and creation of the arrays of zeros, and then plot using `imagesc`, you should get Figure 4.12. Soooooo close<sup>44</sup>. Many of the continuous blocks of land have correctly been assigned a unique identifying number (the different regions of the same color in the figure). But something 'odd' happens in Eurasia, creating those stripes of color when it should be a solid block. It does not help to change the order of the loop (swapping the inner, `lon` loop for the outer, `lat` one) (Figure 4.13) and similar (but different – why?) artifacts arise (plus now one cell in Antarctica has a different color from the rest of the continent).

The way to debug this problem and write the code needed to adjust the algorithm is to again, work though in your head what happens when the loop is passing over the top of Eurasia. For instance, you can see that the first, mid-blue (value 4 in the `land_id` array) row is correct. But when the next row starts, because it starts at a lower longitude with ocean to the North, simply looking to the W and to the N does not reveal the existence of the row of 4s that start slightly later (in longitude).

As ever, there are a number of (equally correct) ways of correcting this. Here, we'll take the approach of post-processing the array, i.e. we'll leave the code that generates Figure 4.12 alone, but go back through the `land_id` array in a new nested loop, and fix the accidental partitioning of Eurasia into differently numbered strips. One possible solution is given below:

```
for lat=2:n_lat
    for lon=2:n_lon
        if ( (land_id(lat,lon)>0) && (land_id(lat-1,lon)>0)
            ...
                && (land_id(lat,lon) ~= land_id(lat-1,lon))
        )
            old_id = land_id(lat,lon);
            new_id = land_id(lat-1,lon);
            land_id(find(land_id(:,:)==old_id)) = new_id;
        end
    end
end
```

In this, we skip the first row (Northern-most latitude) and first column (Western-most longitude) completely, because one might suspect that these grid points cannot be incorrectly labelled (why?), hence the `2:n_lat` and `2:n_lon` loop limits. The issue we are having and why the previous algorithm did not fully succeed, is that some of the land masses have been split into sperate strips, where adjacent cells sharing the same longitude, have different index values. i.e. we need to look for grid cells which have a different index value to the cell immediately to the North, as long as neither is ocean

<sup>44</sup> Note that one could also question the decision to not count diagonal connections as representing continuous land. The result is that the single cell representing Spain and Portugal, is assigned a unique identifier. However, allowing diagonal connections would have the effect of joining North and South America.

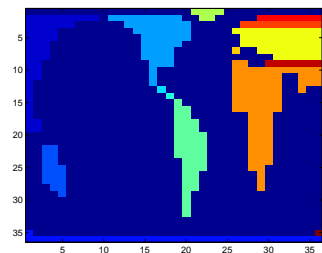


Figure 4.13: The 'GENIE' mode land grid, with land points assigned a unique identifier (color).

(0). The way I have structured the if statement is to test for both `lat` and `lat-1` cells not being 0, AND the two cells not being equal (i.e. having a different value). The result of applying this corrector code is shown in Figure 4.14.

Finally ... the longitudinal edge of the domain is also creating a problem, and land, which should be continuous across the longitudinal domain boundary is instead treated as separated (i.e. the Eastern edge of Eurasia on the LH edge of the plot is one color, but the rest of Eurasia (RH side) is another ... We can fix this by adding one further correction:

```
for lat=1:n_lat
    if ( (land_id(lat,1)>0) && (land_id(lat,n_lon)>0)
    ...
        && (land_id(lat,1) ~= land_id(lat,n_lon)) )
        old_id = land_id(lat,n_lon);
        new_id = land_id(lat,1);
        land_id(find(land_id(:,:)==old_id)) = new_id;
    end
end
```

which works though all the rows (latitude) and checks to see whether the cell in the 1st column has a different value to the one in the last (but with neither being zero) and then makes a substitution of all occurrence of the superfluous label for the correct one, as before. The result of applying this last adjustment to the code is shown in Figure 4.15 and now represents a complete solution to the problem.

Actually ... it doesn't quite represent the final word and if you were a perfectionist, there is one last step to take. If you inspect the contents of the index array you will see that some of the possible values have been skipped<sup>45</sup>. The problem left for the reader (i.e. you) is to re-number the land masses such that for  $n$  land masses, they are numbered from 1 to  $n$ .<sup>46</sup>

This entire example actually took more trial-and-error than I have owned up to. This is no 'bad' thing *per se* and the creation of algorithms for solving problems invariably involves adjustment and refinement of an initial attempt, and sometimes throwing it all away and trying something completely different instead. the key step is to get started and formulate a basic structure for the code and approach. Thus you refine things partly through working through some simple cases to explore what the code really does. Remember – to really test the code you may need to invent cases that don't actually exist in a particular data set in order to put your algorithm through its paces.

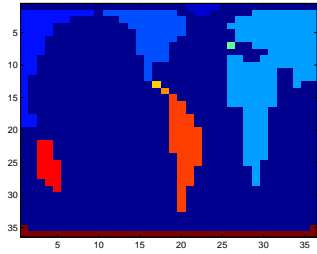


Figure 4.14: The 'GENIE' mode land grid, with land points (almost) assigned a unique identifier (color).

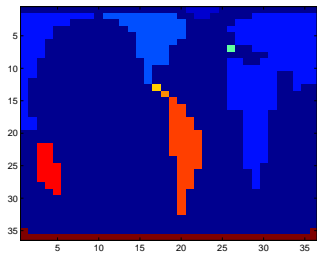


Figure 4.15: The 'GENIE' mode land grid, with land points assigned a unique identifier (color).

<sup>45</sup> Because we re-numbered them earlier, right?

<sup>46</sup> HINT: You could find the highest land mass index value, loop through these values, and for each missing value that is found, renumber the next existing value to the missing one. Or something like that.

170 str = 'do you like bananas?'

### 4.3 *Interpreting equations (o) – Basics*

## 4.4 Interpreting equations (1) – Population models

### 4.4.1 Exponential (and unrestricted) growth

CONSIDER THE SIMPLE MATHEMATICAL POPULATION MODEL<sup>47</sup>:

$$P_{(n+1)} = \lambda \cdot P_{(n)}$$

This defines the number of individuals in the population that there will be at some point in the near future, based on the number at the current time, where:

- $P_{(n)}$  ... is the size of the population at generation (or time)  $n$ .
- $P_{(n+1)}$  ... is the size of the population at generation  $n + 1$ .
- $\lambda$  ... is the average number of offspring produced, per adult per generation, less mortality.

Don't get put off by all the  $N$ s and *subscripts* and things. All the equation says is that the population size (number of individuals =  $P$ ) at the time of the next generation ( $n + 1$ ) is equal to the population at the current generation ( $n$ ) multiplied by some factor. This factor is given the Greek letter  $\lambda$ .<sup>48</sup> The factor  $\lambda$  includes both gains due to the production of offspring and losses from the population due to snowboarding off of a cliff or some other way of dying or being eaten.

So, we are simply asking; how many individuals will there be at the time of the next generation ( $n + 1$ )? The answer is; the same number as currently, minus the fraction of the population who snowboard off of a cliff or die of old age,  $\alpha P_{(n)}$ , plus the number of births in the population, which is also assumed proportional to the current number of individuals in the population,  $\beta P_{(n)}$ .

If there are  $P$  individuals in the current generation, the number at the next generation can be written:

$$P_{(n+1)} = P_{(n)} + \beta \cdot P_{(n)} - \alpha \cdot P_{(n)}$$

In code, this would look like:

```
P = P + beta*P - alpha*P;
```

Re-arranging, we get:

$$P_{(n+1)} = (1 + \beta - \alpha) \cdot P_{(n)}$$

The only even faintly subversive thing that has happened to the original equation, is that all these factors have been included in the value of  $\lambda = (1 + \beta - \alpha)$ .

Simple, eh? Mostly, that is about all there is to computer modelling. You know how much stuff (rabbits, snowboarders, cloud

<sup>47</sup> Modelling animal and plant populations using simple equations gives insights to the population dynamics (i.e. whether numbers remain stable, or go up and down slightly from year to year, or oscillate up and down wildly - almost to extinction one year and increasing to pest levels the next).

<sup>48</sup> We could equally write this in terms of time and if the units of  $\lambda$  were per year ( $\text{yr}^{-1}$ ), rather than generation number  $n$  we would have time  $t$  (years since the start (of the model)).

water droplets, whatever) there is currently (or at a specific point in time), and you want to predict how much there will be in the future, which you take to be one unit of time (time-step) away. You estimate the change in quantity (rabbits, snowboarders, cloud water droplets) that occurs over the course of one generation, and add it to the current quantity.

This model predicts that as long as  $\lambda > 1$ , the population will increase exponentially, generation by generation, without end. Think of bacterial cells dividing in a petri dish. On each subsequent generation (or time step) there will be twice as many cells as there are currently (assuming that all the cells divide into two at the same rate and there is no mortality of cells). The value of  $\lambda$  in this example would be 2.

So to kick off – create a model of this system. You are going to need a (single) *loop* – your choice as to whether you fix the number of iterations (time-steps) beforehand in a `for` loop, e.g.

```
for ...
    P = P + beta*P - alpha*P;
end
```

or use a `while ... end` construction and ensure the expression evaluates to *false* when a set number of cycles of the loop is reached (you'll need to create a counter for this), or the model might end when a certain degree of convergence (on a solution) has been achieved – i.e. when from time-step to time-step, the change gets smaller and smaller each time and at some point gets smaller than some pre-determined threshold.<sup>49</sup> You might use a variable to govern how many iterations are executed (however you do this) rather than hard-code in a value. The value of this variable could be set near the start of the code, or the m-file could be configured with the number of iterations passed in as an input parameter. You'll also need to specify the initial value of the population.

You'll probably want to plot the results<sup>50</sup> and so you may want to save the data of population number vs. generation or iteration (i.e. 2 columns of data and a number of rows equal to the number of iterations through the loop plus one (why?)). The `save` function can be used for this.

#### 4.4.2 Restricted growth (and an equilibrium state)

IN A VARIANT OF THIS ... one might consider that most plant or animal (or bacterial or snowboarder) populations do not behave like this – instead they vary around some average level. This is because birth & death rates vary depending on the size of the population. For example:

<sup>49</sup> This of course rather depends on the solution converging and not oscillating or exponentially growing ...

<sup>50</sup> Your choice of a linear or log y-axis scale – use the one that enables the most information to be presented and in the most useful way



- When the population is large, there may be little food to go round and the birth rate falls (or death rate increases).
- Or, when the population is very small, all individuals may have access to as much food as they can eat giving a high birth rate (or low death rate). For the bacteria in a petri dish, the population cannot go on expanding for ever – sooner or later the entire surface of the nutrient agar will be covered, leaving no free space for new cells to sit happily directly on the food. Later, the nutrients in the agar might start to become depleted. Toxic waste products might also start to build up, slowing down the rate of growth and cell doubling in the bacteria.

We can include a density-dependence by modifying the original equation, to give:

$$P_{(n+1)} = \frac{\lambda \cdot P_{(n)}}{(1+a \cdot P_n)^b}$$

There are two new parameters here:

- $b$  ... defines the strength of the density dependence and the dynamics of the population, and
- $a$  ... is a scaling factor.

Try starting with values of:

- $\lambda = 2.0$
- $b = 0.1$
- $a = 0.1$

and run for e.g. 100 or 1000 generations (or however you are counting the loop in units of). Then systematically investigate the effect of changing the value of parameter  $b$  on the dynamics of the population, keeping the values of the parameters  $\lambda$  and  $a$  constant.<sup>51</sup> Increase the value of the parameter  $b$  and investigate how the dynamics change. Try values of  $b$  in the range 0.1 to 10. Try and find the approximate range of values of  $b$  that give the following types of dynamic of the population:

1. **Monotonic Damping** (smooth approach to a stable equilibrium).
2. **Damped Oscillations** (oscillates to start with then dampens down to an equilibrium).
3. **Stable Limit Cycles** (regular pattern of peaks and troughs with the population repeatedly returning to exactly the same size).
4. **Chaos** (population bombs about all over the place with no regular pattern).<sup>52</sup>

<sup>51</sup> This sort of exercise is known as a sensitivity analysis – i.e. quantifying the sensitivity of the model behavior or final result, to the value of a particular parameter.

<sup>52</sup> Actually, some of the behaviour of population size in the model is probably not real – for certain ranges of parameter value, the model is no longer numerically stable. It is this that gives rise to some of the strange population size behaviour.

Don't spend too much time playing. I know how much fun you are having ;) The key take-home message is to recognise that the population value at each subsequent generation or iteration ( $n + 1$ ) depends directly on the value at the previous one ( $n$ ).

Here you are using a numerical model to explore how a system behaves, and how sensitive the behaviour is to a critical parameter ( $b$  in this example). This sort of exploratory investigation can help you identify critical parameter values that have a profound (and maybe unexpected) effect – for instance, if parameter  $b$  related to something that was impacted by climate change, you might be able to determine the point in the future when climate change might make a population unstable. You might identify a certain population level as genetically viable (anything below this being un-viable). You might then be in a position to make recommendations about conserving this species. And all from just playing around with a computer model!

## 4.5 Interpreting equations (2) – Pure lovely maths

HERE, we are going to code up a graphical representation of the Mandelbrot Set – Figure 4.16, Figure 4.17, and see Box. But we are going to do this nice and gently, via a simplified example.

### 4.5.1 Sequence convergence (in 1D)

CONSIDER the mathematical sequence:

$$z_{(n+1)} = z_{(n)}^2 + c$$

Here – each successive,  $(n + 1)$ -th value of  $z$ , is equal to the  $n$ -th value of  $z$  squared, plus  $c$ . We would write this in code:

```
for n=1:n_max
    z = z^2 + c;
end
```

where the new value of  $z$  is set equal to the previous value squared, plus the value of  $c$ . For the code to work – missing so far here is the initial value of variable  $z$ , as well as what variable  $c$  is.

We'll start the value for  $z$  of zero, and the code<sup>53</sup> would look like:

```
n_max=10;
z = 0;
for n=1:n_max
    z = z^2 + c;
end
```

Here, defining beforehand (at the start of the code) the number of iterations ( $n\_max$ ) that the loop will go through.

We are interested in whether, for a given value of  $c$ , the value of  $z$  grows ever larger and larger (without limit for ever), or whether it settles down and converges on some (finite) value.

You can hopefully see by inspection of the code (and/or equation), and trying out different values of  $c$ , that some values of  $c$  lead to the iteration converging, or remaining finite and small, while others lead to progressively larger values, apparently growing without limit. For some example (real number) values of  $c$  and the sequences of  $z$  they lead to, refer to Table 4.1.

We could sort through a range of values of  $c$ , and for each value of  $c$ , apply the equation:

$$z_{(n+1)} = z_{(n)}^2 + c$$

iteratively, carrying out a given maximum number of iterations. We could then determine for which values of  $c$  the sequence converges, and for which it does not, e.g. <sup>54</sup>

The **Mandelbrot Set**, is the set of complex numbers  $c$ , for which:

$$\lim_{n \rightarrow \infty} |z_{(n)}| \leq 2$$

where

$$z_{(n+1)} = z_{(n)}^2 + c$$

and

$$z_{(0)} = 0$$

which ... shares all of the characteristics of gobbledygook, and I probably haven't even defined it mathematically correctly ...

A rendition of the solution is shown in Figure 4.16 and zoomed-in, in Figure 4.17.

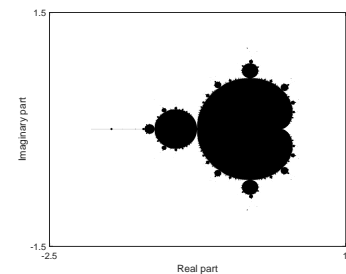


Figure 4.16: The Mandelbrot Set – points representing complex numbers that are members of the set, are shown in black. Complex numbers for which the sequence does not converge, are graphically represented by the white locations in the plotted domain.

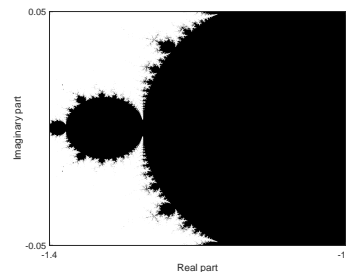


Figure 4.17:  $\times 50$  (-ish) zoom in on the Mandelbrot Set illustrating self-similarity and the fractal nature of the set boundary.

<sup>53</sup> Assuming just 10 iterations of the sequence.

<sup>54</sup> Instead of writing

$$z = z^2 + c;$$

faster is:

$$z = z*z + c;$$

value of $c$	sequence of values of $z$ , as $n$ increases (starting at $n=0$ )
-3.0	0.0 → -3.0 → 6.0 → 33.0 → 1086.0 → ...
-2.0	0.0 → -2.0 → 2.0 → 2.0 → 2.0 → ...
-1.0	0.0 → -1.0 → 0.0 → -1.0 → 0.0 → ...
-0.5	0.0 → -0.5 → -0.25 → -0.4375 → -0.30859375 → ...
0.0	0.0 → 0.0 → 0.0 → 0.0 → 0.0 → ...
0.5	0.0 → 0.5 → 0.75 → 1.0625 → 1.62890625 → ...
1.0	0.0 → 1.0 → 2.0 → 5.0 → 16.0 → ...
2.0	0.0 → 2.0 → 6.0 → 38.0 → 1444.0 → ...
3.0	0.0 → 3.0 → 12.0 → 147.0 → 21612.0 → ...

```

% clear workspace and close open figures
clear all;
close all;
% set (maximum) number of iterations to carry out
n_max=10;
% create sequence of numbers to test (vector)
v = [-3:0.1:3.0];
% fetch number of numbers in sequence (vector length)
n_v = length(v);
% loop through all the numbers in the sequence
for m=1:n_v
    % initialize (zero) value of z
    z = 0;
    % set value of c from vector
    c = v(m);
    % loop
    for n=1:n_max
        z = z^2 + c;
    end
    % assign result value depending on whether converged
    if (z > 2)
        v_conv(m) = 0;
    else
        v_conv(m) = 1;
    end
end
end

```

Here, after  $n = 10$  iterations, the code tests whether the value of  $z$  has exceeded 2.0<sup>55</sup> If the value of  $z$  has surpassed this threshold by the end of the 10 iterations, we are assuming that the sequence will never converge for this particular value of  $c$ . If not converging, the value of the vector  $v\_conv$  (at the same index as the value of  $c$  was extracted from), is set to 0, otherwise, 1.

We could visualize the values of  $c$  for which it converges, via:

```

figure;
axis([-3 3 -1 1]);
scatter(v,zeros(1,n_v),50,v_conv,'filled');
xlabel('Value of c');
ylabel('n/a');

```

where I have created a dummy  $y$ -axis, with dummy (zero values). Each point is large and filled and colored according to the value contained in  $v\_conv$ , which is either 1 (converging) or 0 (not converging after 10 iterations). The result is shown in Figure 4.18.

Table 4.1: Examples of applying the equation iteratively (different starting values).

<sup>55</sup> Here, we are assuming rather simplistically, that 2.0 is a reasonable threshold for testing for convergence. It is somewhat arbitrary and a different criteria could equally have been used.

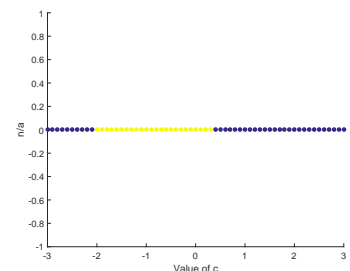


Figure 4.18: Solution space (blue points) for the simple sequence.

We can make the plot a little more interesting, by color-coding a measure of how quickly the sequence accelerates away to increasingly high values. For instance, we could color the points, if not converging, as some function of the highest value of  $z$  reached (when  $n = 10$ ), e.g.

```
if (z > 2)
  v_conv(n) = 1/z;
else
  v_conv(n) = 1;
end
```

It turns out this is not very effective, as after 10 iterations, if not converging, generally very large values have been reached, and so in the color scheme, all non-converging values of `v_conv` are still close to zero.<sup>56</sup>

An alternative, is that for any given value of  $c$ , we identify how many iterations it takes to surpass the prescribed threshold (2.0) – the faster the sequence diverges, the fewer iterations of the loop will be needed to surpass the threshold. Now we cannot simply loop from 1 to 10 using a fixed `do` loop, because the value of 2.0 might be exceeded long before 10 iterations total has been reached.<sup>57</sup>

Instead, we could use `while`. A basic substitution of the current inner (`do`) loop would look like:

```
n = 0;
while (n <= n_max)
  z = z^2 + c;
  n = n+1;
end
```

Try this and satisfy yourself that it does exactly the same as before.

So now we add the additional criteria for terminating the `while` loop also test for the threshold being surpassed:

```
n = 0;
while ( (n <= n_max) && (z <= 2) )
  z = z^2 + c;
  n = n+1;
end
```

Now, the loop continues only if there are more allowed iterations (`n_max` has not been reached yet), and, the threshold has not yet been exceeded.

This code is faster than before, but your problem is pretty simple and you may not notice.<sup>58</sup>

The final step is to take the value of  $n$  that is reached when the `while` loop terminates, and use that to plot the color-scale.

<sup>56</sup> Nor does it help to simply set:

```
if (z > 2)
  v_conv(n) = z;
else
  v_conv(n) = 1;
end
```

(Try it and see!)

<sup>57</sup> Actually, we could use a fixed `do` loop, but it is much more efficient not to – if early on in the loop, the threshold has been surpassed, why keep iterating (and wasting CPU cycles)?

<sup>58</sup> If you would like to explore the efficiency of your program a little further:

1. At the very start of the code, add the line:  
`tic;`  
and at the end of the program, add:  
`toc;`  
Giving you a timing of the code execution.
2. Comment out all the lines of code for the graphics, so that you are left only with the calculations (and initialization).
3. Force the program to carry out a more challenging number of calculations, e.g.

```
v = [-3:0.000001:3.0];
n_max=100;
```

178 str = 'do you like bananas?'

```
...
% loop
n = 0
while ( (n <= n_max) && (z <= 2) )
    z = z^2 + c;
    n = n+1;
end
% assign value depending on whether converged
if (n == n_max)
    v_conv(n) = 0;
else
    v_conv(n) = 1.0-n/n_max;
end
...
```

Here, the test for convergence of the sequence is out counting variable  $n$  having reached a value of  $n\_max$  (i.e. for all the maximum allowed iterations of the loop, the value of  $z$  has remained less than or equal to 2.0).

Potting this now, looks like Figure 4.19.

#### 4.5.2 Sequence convergence (in 2D)

Now to the Mandelbrot set ...

The idea is basically the same as before – we are going to generate a sequence, and find out whether it converges, or if not, how quickly it diverges and whizzes off towards infinity in value. The equation is very similar to before (see Box), with the next value equal to the current value squared, plus a constant, and we are varying the value of the constant.

The big complication is that  $c$ , is now not a simple *real* number (and one that we could simply plot along the  $x$ -axis), but a *complex* number (see Box).

It is helpful to think of the real and imaginary components of the number, as  $x$  and  $y$  values on an  $x$ - $y$  plot<sup>59</sup> and treat them exactly as per you would vectors.

How to put this into code?

Well, the number  $c$  now has two parts – a *real* an *imaginary* part. Lets call them variables  $x$  (*real*) and  $y$  (*imaginary*).

The number  $z$  also has two parts. We could represent these by variables  $a$  and  $b$ . If we simply had the equation:

$$z_{n+1} = z(n) + c$$

we could write (within a loop):

```
a = a + x;
b = b + y;
```

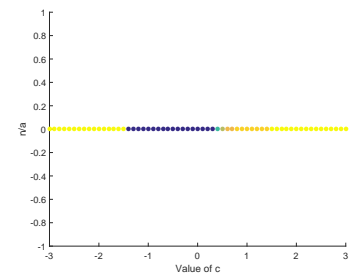


Figure 4.19: Solution space (blue points) for the simple sequence, with the rate of divergence forming the color scale of light blue (slowest) through yellow (fastest divergence).

A *complex* number  $z$ , is a number of the form:

$$z = a + bi$$

where  $i$  is the square root of  $-1$  (or  $i : i^2 = -1$ ).

If we square  $z$ , we have:

$$\begin{aligned} z^2 &= (a + bi) \times (a + bi) \\ &= a^2 + a \times (bi) + (bi) \times a + (bi)^2 \\ &= a^2 + 2 \times a \times b \times i + (b)^2 \times (i)^2 \end{aligned}$$

and remembering what  $i^2$  equates to:

$$z^2 = a^2 - b^2 + 2 \times a \times (bi)$$

<sup>59</sup> And in fact, this is exactly how we will be plotting things later.

For the equation

$$z_{n+1} = z_{(n)}^2 + c$$

we now have:

$$\begin{aligned} a &= a^2 - b^2 + x; \\ b &= 2*a*b + y; \end{aligned}$$

(see Box). Except ... although we have taken the value of  $a$ , updated it, reassigned it back to the variable  $a$  ... when it comes to updating the value of  $b$  ... whoops(!) – we have already updated  $a$  (and we should not have as the original value is needed to calculate  $b$ ). The simplest solution is to make the old and new values completely explicit:

```
a_old = a;
b_old = a;
a = a_old^2 - b_old^2 + x;
b = 2*a_old*b_old + y;
```

The equation above ( $z_{n+1} = z_{(n)} + c$ ) in code, for  $n_{\max}$  iterations (of  $n$ ), looks like:

```
do n=1:n_max,
    a_old = a;
    b_old = a;
    a = a_old^2 - b_old^2 + x;
    b = 2*a_old*b_old + y;
end
```

Again, we could replace the `do` with a `while` as as before, apply a convergence criteria to terminate the loop (early):

```
while ( (n <= n_max) && ((a^2+b^2) <= 2^2) )
    a_old = a;
    b_old = a;
    a = a_old^2 - b_old^2 + x;
    b = 2*a_old*b_old + y;
end
```

or faster (but probably not important in this particular **MATLAB** program) would be:

```
while ( (n <= n_max) && ((a*a+b*b) <= 2*2) )
```

(because multiplication is faster for computers than raising a number to a power).

Mathematically, that's it. What remains is to create a set of values of  $c$  to test for convergence on, and because *complex* numbers can be represented in  $x$ - $y$  space, we can create a 2D grid of real and imaginary component values, just as we did early for lon-lat values in plotting maps.

```
180 str = 'do you like bananas?'
```

For example:

```
x = [-3:0.1:3.0];  
y = [-3:0.1:3.0];
```

would create a range of *real* and a range of *imaginary* parts of the *complex* number  $c = x + yi$ .

However, as per for lon-lat, we want all combinations in a 2D grid, and so we use `meshgrid`:

```
[xx, yy] = meshgrid([-3:0.1:3.0],[-3:0.1:3.0]);
```

---

At this point – pause.

1. You have the loop framework code to test whether the maximum number of iterations has been reached, or whether the test of convergence has failed.
2. You have the code in the iteration loop, to square one complex number ( $z$ ) and add a second ( $c$ ) to it.
3. You have create a pair of matrices – one of values of  $a$  ( $xx$ ) and one of  $b$  ( $yy$ ) which together, map out a 2D space (/grid) to be searched.

Next, the full code will be provided to you (as an alternative to you trying to piece fragments together), but it is your job to make sure you understand it ...



```

% clear workspace & close open figure windows
clear all;
close all;
% create a parameter to contain the threshold value
thresh = 2*2;
% maximum number of iterations
n_max = 10;
% create initial grid ...
% from -1 to +3 in both dimensions, ...
% with a step resolution of 0.1
[xx,yy] = meshgrid([-3:0.1:3.0],[-3:0.1:3.0]);
% determine total number of points to test
m_max = numel(xx);
% reshape x and y matrices into 2 columns of vectors
v(:,1) = reshape(xx,[m_max,1]);
v(:,2) = reshape(yy,[m_max,1]);
% create a 3rd vector column ...
% for storing a measure of convergence/divergence
v(:,3) = zeros(m_max,1);
% loop through the x-y vector columns
for m=1:m_max
% set the value of complex number c
    x = v(m,1);
    y = v(m,2);
    % initialize z(n=0)
    a = 0.0;
    b = 0.0;
    % initialize the count
    n = 0;
    % iterate and check for convergence
    while ( (n <= n_max) && ((a*a + b*b) < thresh) ),
        % copy old value of z (n)
        a_tmp = a;
        b_tmp = b;
        % update z (n+1)
        a = a_tmp*a_tmp - b_tmp*b_tmp + x;
        b = 2*a_tmp*b_tmp + y;
        % update count
        n = n+1;
    end
    % set measure of convergence/divergence
    if (n <= n_max),
        v(m,3) = 1.0/n;
    else
        v(m,3) = 0.0;
    end
end
end
% take results vector, and ...
% reshape back into matrix form (for plotting)
zz = reshape(v(:,3),[length([-3:0.1:3.0]),length([-3:0.1:3.0])]);

```

```
182 str = 'do you like bananas?'
```

In this code, you should note that I have avoided a double/nested loop for looping through the 2D space of the real ( $a$ ) and imaginary ( $b$ ) parts of the complex number  $c$ . Instead, I have simplified this to a single loop, of all elements. The total number of elements in the grid can be obtained using the `numel` function<sup>60</sup>.

Knowing the total number of elements in the `xx` and `yy` matrices, it is a simple matter to convert these into vector form:

```
v(:,1) = reshape(xx,[m_max,1]);  
v(:,2) = reshape(yy,[m_max,1]);
```

and then to add a 3rd column, that will hold the results:

```
v(:,3) = zeros(m_max,1);
```

---

You can plot the resulting grid of convergence/divergence values using `imagesc`:

```
imagesc(zz);
```

as per Figure 4.20.

To obtain a higher resolution plot – simply increase the resolution of the  $x$  and  $y$  vectors used by the `meshgrid`<sup>61</sup>. Also – the maximum of iterations allowed, `n_max`.

To create the simple black/white plot (e.g. Figure 4.16), I created a color scale which is all white, apart from black at the very start of the scale (corresponding to the lowest values). To do this, near the start of the code (before any of the loops), or, after both loops have finished, add:

```
% create color scale  
brotmap = [ 0 0 0;  
           1+zeros(9,3)];
```

and which looks like:

```
» brotmap  
brotmap =  
    0 0 0  
    1 1 1  
    1 1 1  
    1 1 1  
    1 1 1  
    1 1 1  
    1 1 1  
    1 1 1  
    1 1 1  
    1 1 1  
    1 1 1  
    1 1 1
```

<sup>60</sup> In the code, it does not matter whether you write:

```
m_max = numel(xx);
```

or

```
m_max = numel(yy);
```

as the 2 matrices are exactly the same size.

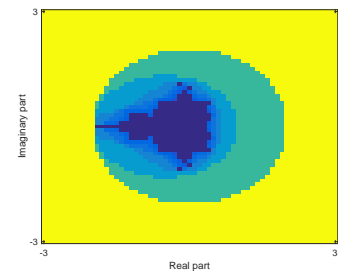


Figure 4.20: Simple, low resolution Mandelbrot set rendition.

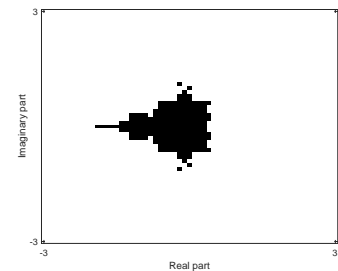


Figure 4.21: Simple, low resolution Mandelbrot set rendition (now highlighting points that are members of the solution set (black) vs. not (white).

<sup>61</sup> Note that you also have to make the same changes at the end when you reshape the results to the matrix `zz`.

defining black on the first row, and white on the next 9 rows. This gives a color scale of ten rows, that corresponds to the maximum number of iterations in your code, and hence the maximum number of different values that `zz` can take.<sup>62</sup>

Then, before you call `imagesc`, add:

```
cmap = colormap(brotmap);
```

The result is shown in Figure 4.21.

A more flexible way to define the grid limits and the resolution, is instead of writing in directly the specifications passed to `meshgrid`, e.g.:

```
[xx, yy] = meshgrid([-3:0.1:3.0],[-3:0.1:3.0]);
```

is to first set the grid limits:

```
x_min = -2.5; x_max = 1.0; y_min = -1.5; y_max = 1.5;
```

define the resolution – here the number of divisions:

```
xy_res = 2000;
```

and then for `meshgrid`:

```
[xx, yy] = meshgrid ...
    ([x_min:(x_max-x_min)/xy_res:x_max], ...
    [y_min:(y_max-y_min)/xy_res:y_max]);
```

This particular line is ‘messier’ than before, but now it is much easier to change the grid limits, and/or the resolution, and when you convert the results vector to a matrix, it is now just:

```
zz = reshape(v(:,3),[xy_res+1,xy_res+1]);
```

Try playing about with Mandelbrot Set plots – changing the  $x$ - and  $y$ -limits (`x_min,x_max,y_min,y_max`) as well as the resolution (`xy_res`) of the plot.

Figures 4.22, 4.23, 4.24 give examples of different regions (zooms)..

These example plots also employ a slightly more complicated color scheme:

```
brotmap2 = [ 0 0 0;
            jet;
            flipud(jet)];
```

which defines, as before, black as the color corresponding to the lowest values – in this case the solution set (a sequence that converges). But then it adds the built-in **MATLAB** `jet` color scheme to the end of this. And then ... for good measure, it adds on another copy of

<sup>62</sup> Other choices for number of rows would have been perfectly acceptable in this particular example.

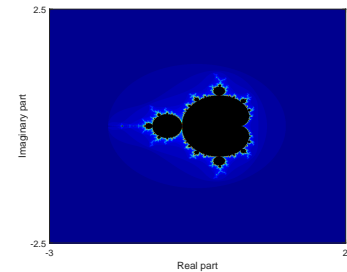


Figure 4.22: Initial Mandelbrot Set magnification.

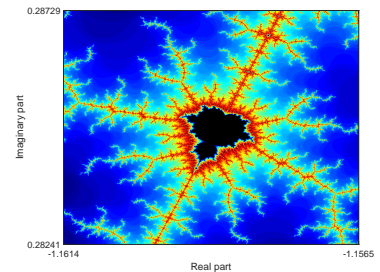


Figure 4.23: Example Mandelbrot Set zoom.

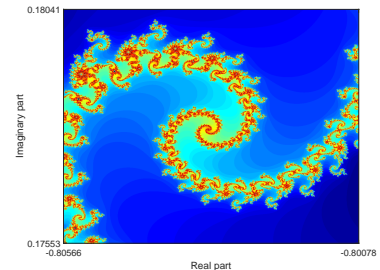


Figure 4.24: Example Mandelbrot Set zoom.

jet, but this time inverted<sup>63</sup> (colors occurring in the opposite sequence).

<sup>63</sup> The `flipud` accomplishes this inversion.

---

The real advantage of defining  $x$ - $y$  limits it this way, is that you can re-formulate the code as a *function*, get the position of the mouse on the screen, and click to zoom by some fixed and predetermined amount, or to define a box to zoom to, and pass the new, updated  $x$ - $y$  limits back to the *function* to re-calculate the sequence, and then re-plot the now zoom-ed in region of solution space.

In re-formulating your script as a function – you take as input, the  $x$ - and  $y$ -limits (four variables total), and return as output, an array of results values (`zz`). You would call this function from a *script* that does the actual plotting of the figure. The script would also set initial (default)  $x$ - and  $y$ -limit values and ... once the figure is drawn, take mouse input for a single click (to define the center of a zoomed in region) or two mouse clicks (to define the opposite corners of a zoomed in region) and then re-draw the plot.

Example code of a *script* ([make\\_brot.m](#)), and the corresponding *function* ([fun\\_brot.m](#)), are provided via the links.<sup>64</sup>

<sup>64</sup> Simply type:

» `make_brot`

to start, left mouse-button click to zoom to that point, and right-button mouse click to end (the plot window remains open however).

Zoom is controlled by the parameter `xy_mag` in `make_brot.m`.

## 5

### *Programming applications – games!*

GAMES are great examples of many of the different facets of computer programming and **MATLAB** covered to date. They invariably contain algorithms and require problem-solving in the code, will contain multiple functions and sub-programs, loops, conditionals, graphics of some sort. They will invariably be complex, and hence put debugging skills to the test. They often even contain physics (and science)! They also provide an important motivation for developing the code – a specific and hopefully fun, end-product.

## 5.1 Tic-tac-toe

TIC-TAC-TOE<sup>1</sup> – Figure 5.1 – ‘is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a  $3 \times 3$  grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.’ It is as common a muck and inevitably, everypony knows how to play it, so we don’t need to spend time defining the rules.

Here we’ll devise a basic version requiring 2 (mostly) human players, but note the possibility of developing an AI computer player(!) We’ll also not make use of the **MATLAB** GUI, so keep the code as simple as possible, but note that as a further possibility for development.

**IMPORTANT:** In the sections that follow – a number of code fragments are given to you. The idea is not simply to copy-paste the code fragments and go home. The key is getting the structure of the program (and how the various sperate functions are created and utilized) right. If you find yourself having no idea ‘where’ to put a particular code fragment ... you have probably not understood how the program is constructed and are simply going through copy-paste motions. Note that you can always simply write your own code from scratch – there are many (infinite?) different ways of creating the program and writing the code to solve the different steps. You might even find that easier as it should be more obvious to you ‘where’ to put different lines.

A schematic of the complete (final) code structure is shown in Figure 5.2 as a guide.

The following is a brain-dump on what we need to go about designing and writing the game<sup>2,3</sup>:

1. The game is almost wholly visual and so we need to think about the graphics at the outset. For instance:
  - We need a Figure window (no really)! Not having any axes/axes labels showing would be nice.
  - We need to draw a grid consisting of two pairs of lines, at 90 degrees to each other.
  - We then need to add a cross or a zero to the graphic when a player clicks an empty square.
  - If there is a winner, we need to draw a line through the winning diagonal or row/column and the game finishes.
2. Associated with the adding of a cross or zero – we need to find a way of a identify which box a player chooses, and apply the cross/zero according to the selected box and player identity.

<sup>1</sup> Also called ‘Noughts and crosses’.

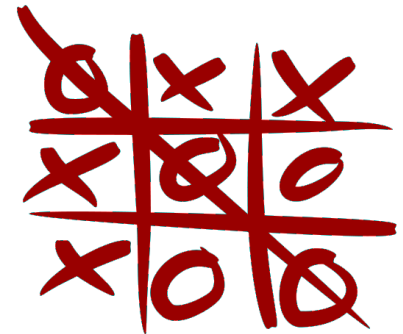


Figure 5.1: Tic-tac-toe. By Symodeo9 - Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=2064271>.

<sup>2</sup> You can devise all sorts of strategies for creating the game, but you do need some sort of strategy before you start to write any code.

<sup>3</sup> Despite the messy, additional automatically-generated **MATLAB** code, some of these are actually easier in a GUI.

3. We need to keep a list of empty boxes in the grid and only allow a box to be chosen if it is empty.
4. We need to test for a winning line of crosses or zeros.
5. We also need some directions to be given to the players – who's turn it is, and who wins, or if there is a draw, that the game is over. To simplify things, these messages can be sent to the command line.

Some of these things you have seen and you will know (hopefully) how to go about it, such as drawing the grid, printing messages at the command line. Others are not so obvious and may prove tricky, so we'll tackle those first – to my mind, these are:

1. Selecting a location in the window.
2. Filtering the chosen position to identify a specific box, and hence position the cross/zero neatly.
3. The mechanics of drawing the cross or zero.

Before we go through these and progressively build up a working game, let's write a shell program *script* **m-file** that we can test ideas and code in. A possible way to start off follows:

```
% *****
% *** Tic-Tac-Toe game! ***
% *****

% close currently open windows
close all;
% clear variable space
clear all;

% create a new figure window
figure;
% create a set of invisible axes that will the window
fh = axes('Position',[0 0 1 1],'Visible','off');
% scale the axes
axis([0 3 0 3]);
% hold on!
hold on;
```

This should mostly be self-evident. You need not have `close all` ... but you may not wish to accumulate Figure windows for ever. Because this is a *script* **m-file**, not a *function*, the variables and their values remain in the **MATLAB** workspace even after the program is terminated or finishes. `clear all` simply ensure that someone a variable value from a previous run of the program, doesn't somehow interferer with the next run.<sup>4,5</sup>

The line starting `fh = ...` creates a plotting area with no axes visible, and filling the Figure window area `([0 0 1 1])` in normal-

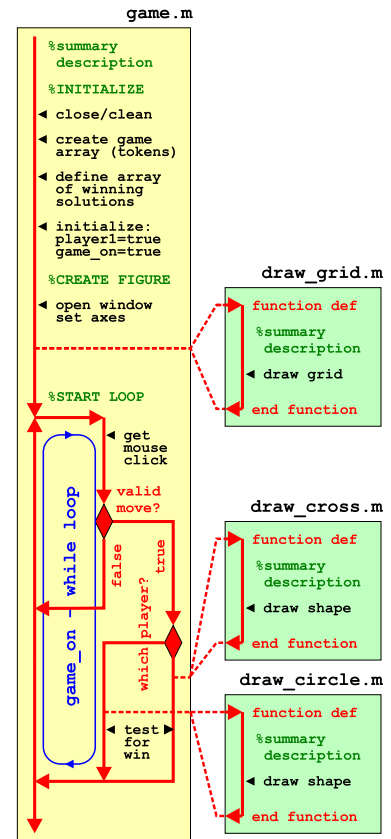


Figure 5.2: Schematic structure of the complete code.

<sup>4</sup> It shouldn't do, and there should be no variables used anywhere, that are not initialized to a specific value first.

<sup>5</sup> If you prefer to frame the program as a *function* ... with no inputs or outputs, then that is fine, but remember that you'll need to add *breakpoints* to interrogate any of the *variable* values as they all become *private*.

```
188 str = 'do you like bananas?'
```

ized units). The handle to this is returned (variable `fh`), just in case we ever need it later.

In scaling the axes – as there are 3 rows and 3 columns in the game area, it seemed ‘reasonable’ to set `axis([0 3 0 3])`, although we need not have.

---

Maybe before getting into any of the listed complexities, we could draw the game grid to give us some visual perspective on things. We could do this perfectly correctly, by adding the code to the main m-file/program file, but it is much neater to put sections of code that do specific things, and particularly if you might want to run these sections of code again, in a *subprogram*, which could be a *script* or a *function*. Here, even though there will be no inputs or outputs, we’ll create a new **m-file function** (to be consistent with additional *functions* that we will be creating) just for the grid drawing – `draw_grid.m`.

This is my *function* for drawing the grid (which will be called from the main program *script* file) and which is saved to an **m-file** `draw_grid.m`:

```
function [] = draw_grid()
%draw the game grid
grid_th = 2.0;
grid_col = [0 0 0];
line([1.0 1.0],[0.0 3.0],'LineWidth',grid_th,'Color',grid_col);
line([2.0 2.0],[0.0 3.0],'LineWidth',grid_th,'Color',grid_col);
line([0.0 3.0],[1.0 1.0],'LineWidth',grid_th,'Color',grid_col);
line([0.0 3.0],[2.0 2.0],'LineWidth',grid_th,'Color',grid_col);
end
```

(You should comment your version better!)

The 4 main lines, simply draw the 4 grid lines – 2 horizontal and 2 vertical. Because the line width and color appear 4 times – one in each `line` command line, I have set the value of a pair of parameters at the start – if I ever want to change line thickness and/or color, I need only make an edit in a single place (where the parameters are defined) rather than in each and every `line` command line. (You could, for instance, experiment with different line thicknesses and colors.)

In your main program (script m-file) – somewhere after `hold on` (refer to Figure 5.2), add the line<sup>6</sup>:

```
draw_grid();
```

which then calls the *function* to draw the grid.

Run it so far. It should look like Figure 5.3, depending on the line width and colors you choose.

<sup>6</sup> Remember you are not passing any parameters to this function, nor is it returning anything back to you

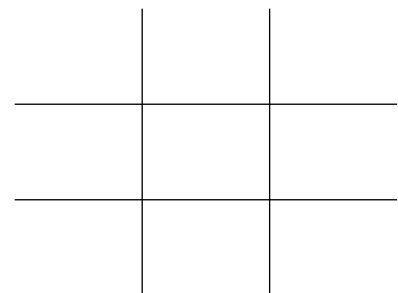


Figure 5.3: Tic-tac-toe game grid drawn.



### 5.1.1 Mouse behavior

OK – so a key part of the game is being able to select a particular grid square (cell), in order to place your (cross or circle) marker. One could do this e.g. at the command line by specifying a coordinate location, e.g. (1,1) for the bottom (or top) left cell, but this would be pretty tedious and would mean flipping back-and-forth between Command Window and Figure Window.

The **MATLAB** function `ginput` is provided to return the coordinate of the mouse pointer when one of the buttons is clicked. The coordinate returned is in the same units as your axes. Nothing is returned if you click outside the Figure Window.<sup>7</sup> `ginput` also returns which of the 2 (or 3) of the buttons was clicked. The `ginput` function also needs to be told how many mouse clicks to return – we need only one (per turn in the game).

To try this out – to the code in `game.m` (and somewhere after `hold on`), add:

```
[x,y,button] = ginput(1);
```

which when you click a mouse button, sets the *variables* `x` and `y` equal to the (`x,y`) location of the mouse button click, and the *variable* `button` to the ID corresponding to which mouse button it was.

If for now, you take the `' ; '` off of the end of the line, and run your program – try clicking the mouse in the game area, and note that hopefully, an (`x,y`) location is reported with values in the range 0 – 3 set by `axis([0 3 0 3])` plus, the ID of which mouse button you used.

So far, so good.

### 5.1.2 Drawing the 'objects'

One strategy in programming, is to get \*something\* happening and working first, and worry about the details and quite what you really wanted, later. So lets draw \*something\* in response to the mouse click, and not yet worry what exactly we draw.

Again – creating subprograms and functions are a great way of reducing clutter in the main program, helpful in debugging, and all but essential if you need to apply the same (or almost the same) code more than once. The need to draw a number of crosses, and also a number of circles, fits the bill. So lets create a pair of functions for drawing first a cross, and then a circle. (See Figure 5.2.)

To begin with, the 2 functions will look almost identical, and differ only in name:

```
function [] = draw_cross(x,y)
%draw cross

end
```

<sup>7</sup> Because we defined the game area as the entire area of the Figure Window, it should not be possible to click in the Figure Window but outside of the game area, so we do not have to deal with that possibility occurring.

`ginput` "ginput raises crosshairs in the current axes to for you to identify points in the figure, positioning the cursor with the mouse ... [x,y,button] = ginput(...) returns the x-coordinates, the y-coordinates, and the button. button is a vector of integers indicating which mouse buttons you pressed (1 for left, 2 for middle, 3 for right)."

So ... basically – you move the mouse and press a button, and **MATLAB** kindly tells you which button you pressed and where (within the axes) you pressed it,

```
190 str = 'do you like bananas?'
```

for the cross (saved as `draw_cross.m`), with the other one:

```
function [] = draw_circle(x,y)
%draw circle

end
```

(saved as `draw_circle.m`).

Both functions take a pair of parameters,  $x$  and  $y$  as inputs, which will be the  $(x,y)$  locations to draw the objects.

We should draw \*something\* (in the `draw_cross` function) to get things going. For now, try adding (in `draw_cross.m`):

```
dz = 0.25;
patch([x-dz x-dz x+dz x+dz],[y-dz y+dz y+dz y-dz],'b');
```

which is obviously not a cross (nor a circle) ... but it'll do for now.

Assuming the  $(x,y)$  location passed into the function is the centre of the object,  $x-dz$  and  $x+dz$  create  $x$ -coordinate vertices symmetrically either side (of  $x$ ), and likewise for the  $y$ -coordinates. Experiment with the value of  $dz$ .

You could test this simply at the command line ...

```
>> figure;
>> axis([0 3 0 3]);
```

and then call the function, passing whatever pairs of coordinates in the range 0 – 3 that you like, e.g.

```
>> draw_cross(0.5,2)
>> draw_cross(1.5,1)
>> draw_cross(2.333,1.333)
```

Create a similar shape for `draw_circle` ... picking a different color and/or a different shape. And then test this *function* (at the command line) also.

---

We should be in good shape at this point – you have a main program that draws the game grid and you have a pair of functions to draw a shape centered on  $(x,y)$ , although so far you have not joined this up and have only tried passing whatever value you fancy into the square/circle plotting *functions*. You also know how to find the  $(x,y)$  coordinates of a mouse button click.

You could test the code further by allowing multiple mouse-clicks and shape drawing – in the main program, after you have drawn the game grid (and after the `hold on`) – replace the temporary line of code you added earlier, i.e.

```
[x,y,button] = ginput(1);
```

with a *while* loop that initially, is endless, and contains the `ginput` function line:

```

game_on = true;
while game_on
    [x,y,button] = ginput(1);
    draw_cross(x,y);
end

```

(but with your own illuminating comments, of course!).

In this: having `game_on = true` without any line that could set the variable to `false`, means that the while loop, `while game_on` loops forever. You'll have to CTRL-C to get out of this (or close the window), but if you click a number of times first, you start to get something that looks like if your luck was otherwise, this could be a 10 million dollar modern art piece (Figure 5.4).

Now ... as an experiment and test of your coding<sup>8</sup>, try making the other shape appear if the other button is clicked – `ginput` returns a value of 1 to the variable `button` if it is the left mouse button, and 3 if it is the right. (See Figure 5.5.) i.e. in the code fragment above, rather than always calling `draw_cross` each and every time a mouse button is clicked – test for which mouse button it is and either draw a cross or a circle (or whatever shape is currently representing 'circle') depending on which it is.

### 5.1.3 Identifying specific boxes

There is still much to do ... but an obvious and significant next step is to place the objects in specific locations – i.e. centered in the box in which the mouse button occurred. So we need to test the values of `x` and `y` (returned by `[x,y,button] = ginput(1)`), and identify a specific grid box (and its indices).<sup>9</sup>

Lets imagine that the grid is counted from the bottom left hand corner, from 0 to 3 in both `x` and `y` directions, so that the first row or column is in the range 0 – 1, the second 1 – 2, and the third 2 – 3. For `x` we could write something like:

```

if (x < 1)
    xi = 1;
elseif (x < 2)
    xi = 2;
else
    xi = 3;
end

```

where `xi` stands for a variable containing the index (or 'integer') `x`-direction, i.e. which one of the 1st, 2nd, or 3rd, column it is.

Write something similar for the `y`-direction, and add both sets of code to your main program immediately after:

```
[x,y,button] = ginput(1);
```

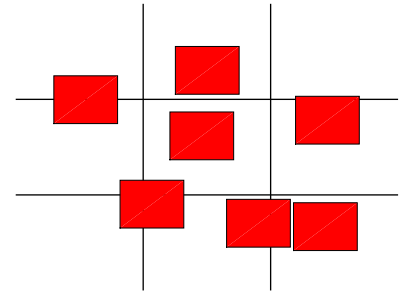


Figure 5.4: Tic-tac-toe game – object drawing test.

<sup>8</sup> If you need a hint – `if ...`

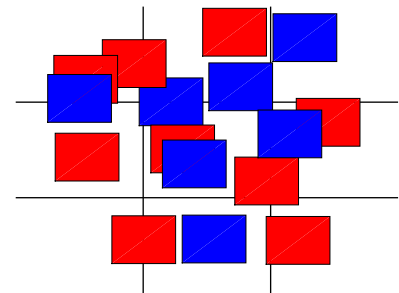


Figure 5.5: Tic-tac-toe game – object drawing + mouse button test.

<sup>9</sup> There is a simpler and sly-er way of doing this, which would be particularly useful if we have a really large grid and having 100s of `elseifs` is not practical. The function `round`, returns the rounded up integer value of a real number. So `round(0.49)` returns 0, while `round` returns 1.

We could derive the value of `xi` simply, in a single line, by:

```
xi = round(x+0.5);
```

where the `+0.5` bit ensures values in the range 0.0 – 1.0 returns 1 (and 1.0 – 2.0, 2).

```
192 str = 'do you like bananas?'
```

so that you determine the  $(x,y)$  location of the mouse click and then convert the  $x$  and  $y$  values into a pair of  $(x_i, y_i)$  indices in the range 1 – 3.<sup>10</sup>

Now, when we call `draw_cross` (or `draw_circle`), we can pass in the  $(x,y)$  coordinates of the center of the square, which will be equal to the value of  $x_i$  (and  $y_i$ ), minus 0.5, i.e.

```
draw_cross(xi-0.5,yi-0.5);
```

Now ... suddenly ... the game seems to be coming together in terms of the graphics (Figure 5.6). (Obviously we are still missing a lot, including correct shapes.)

What you code should look like so far is: after the `hold on` statement in the initial code framework (the very first piece of code given to start building the `game.m` program) is:

```
game_on = true;
while game_on
    [x,y,button] = ginput(1);
    ...
```

immediately followed by one set of code to identify the (integer) column ( $x_i$ ) number:

```
if (x < 1)
    xi = 1;
elseif (x < 2)
    xi = 2;
else
    xi = 3;
end
```

... and then one near identical set of lines of code to identify the (integer) column ( $y_i$ ) number following this:

```
if (y < 1)
    ...
end
```

Then comes the code to determine which mouse button was clicked, and draw a 'cross' or a 'circle' depending on which one it was:

```
if (button == 1)
    ...
end
```

Only then does the `end` of the `while loop` occur in your code.

```
...
end
```

You should be testing this code, which comprises the main (*script*) program `game.m`, plus 3 *function m-files*. It should run for ever, placing a 'cross' in the center of a square if you click with the left

<sup>10</sup> Note that there is an easier way of obtaining the indices, using the **MATLAB** function `round`.

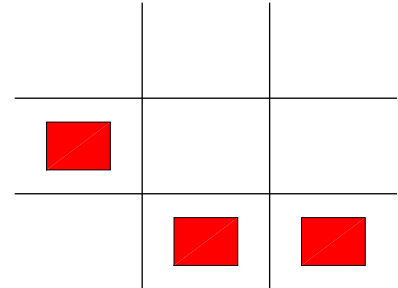


Figure 5.6: Tic-tac-toe game – object drawing now arranged in a grid.

mouse button, and a 'circle' when clicking with the right. Note that currently, you can add multiple squares or circles in a row and there is nothing forcing you to alternative 'turns'. You can also over-write a previously written symbol (in the same square), which is obviously also cheating and something we'll now have to fix.

#### 5.1.4 Remembering turns (and arrays!)

A key to the game is that when a nought/cross has been placed somewhere, you cannot place anything more there. So we need to keep track of which cells have already been chosen. In fact, we need to keep track of what is in each cell.

We will create a  $3 \times 3$  array to store the information in, with each (*row, column*) pair of the **MATLAB** array, corresponding to an (*xi, yi*) pair (cell location in the game grid).

You have already seen how to create an  $n \times m$  array of e.g. zeros. In your program – somewhere near the start, and certainly before the *while loop*, you can add<sup>11</sup>:

```
tokens = zeros(3);
```

(but adding a suitable comment line ...)

Another important thing you'll come across in programming is devising notations for representing states in a model or game or whatever. Pause and think about the possible states that each cell in the game grid can have.

1. Not yet chosen.
2. Assigned to a Cross.
3. Assigned to a Naught.

We could hence decide to assign values in the `tokens` array:

```
0 == Not yet chosen.
1 == Assigned to a Cross.
2 == Assigned to a Naught.
```

(so the elements of the array can take a value of 0, 1, or 2, as illustrated in Figure 5.7).

Now ... in the loop, as the *xi* and *yi* indices are derived in the code – use them to assign a value to the `tokens` array.<sup>12</sup> You can then test the value of the location that has just been chosen, and from this, decide whether the move is legal or not.

In your code, after having determined the values of *xi*, *yi*, and in place of the code to determine which mouse button was clicked, and draw a 'cross' or a 'circle' depending on which one it was, i.e.:

<sup>11</sup> Or call the array variable something better ... there is no completely obvious and helpful variable name for what it will end up holding.

<del>1</del>	2	0
<del>1</del>	2	0
<del>1</del>	0	0

Figure 5.7: Tic-tac-toe game grid with numerical codes overlain.

<sup>12</sup> It is a personal preference whether to simply remember that **MATLAB** indexes arrays differently to reading a normal (*x,y*) location, or try and make the contents of the array, as viewed, look like the game grid.

```
194 str = 'do you like bananas?'
```

```
    if (button == 1)
        ...
    end
```

add the following instead:

```
% test for square is empty (a zero value)
if (tokens(xi,yi) == 0)
    % if empty ... draw cross ...
    draw_cross(xi-0.5,yi-0.5);
    % and then update array that is now has a cross
    tokens(xi,yi) = 1;
else
    % alert if square already taken
    disp('Illegal move! Choose again.');
```

(here setting the value of `tokens` at that location to 1, because we are assuming still the 'cross' player in the variable and function naming notation).

Now when you run your program, when you click in a square, it always draws a cross, but only if there is not already one there.

### 5.1.5 Putting it all together

OK. Pause. Consider where you are at; what you have working ... and what remains to do.

Done:

- Drawn grid.
- Created functions to draw the 2 different game pieces.
- Recovering the  $(x,y)$  mouse click location, and converting that into the game grid  $(x,y)$  location.
- Checking to see whether a game cell is already occupied and not allowing the move if so.

To-do:

- Alternate the player turns.
- Test for the game finishing.
- Draw 'correct' symbols(!)

In terms of player turn – this is a simple binary state – either it is the turn of player #1, or it isn't (and hence the turn of player #2). So we could create a (logical) variable `player1`, to keep track of whose turn it is.<sup>13</sup> If we initialize the game with player #1 starting first, near the top of the main program (and before the `while loop`), we could set:

```
player1 = 1;
```

<sup>13</sup> Equally, we could have defined a variable `player`, that took a value of 1 (for player 1's turn) or 2 (player 2). We would then need to change its value after a player had taken a turn, from 1 to 2, or 2 to 1. This turns out to be more awkward to implement than simply taking the *NOT* of a variable state.

Now it is simple to alternate between the player turns, and after the current player has taken their turn, we can simply write:

```
player1 = ~player1;
```

which flips whose turn it is. This line will go within the while loop and after a player has taken a turn.:

```
if (tokens(xi,yi) == 0),
    draw_cross(xi-0.5,yi-0.5);
    tokens(xi,yi) = 1;
    player1 = ~player1;
else
    disp('Illegal move! Choose again.');
```

At this point, we are still not differentiating between the different players – we need to draw a different symbol depending on which player it is, and also set the corresponding element in the array to a different value (1 for player 1, and 2 for player 2). (Note that we are no longer using different mouse buttons to distinguish player turns ... this was simply a test code that you tried earlier.)

So we need to test for which player it is currently<sup>14</sup>. Now, in place of the 3 lines of code above, i.e.

```
draw_cross(xi-0.5,yi-0.5);
tokens(xi,yi) = 1;
player1 = ~player1;
```

we instead write:

```
if player1,
    draw_cross(xi-0.5,yi-0.5);
    tokens(xi,yi) = 1;
else
    draw_circle(xi-0.5,yi-0.5);
    tokens(xi,yi) = 2;
end
player1 = ~player1;
```

Note that all this goes still within the if test of whether the move is legal or not, i.e.:

```
if (tokens(xi,yi) == 0),
    ...
end
```

If you test the code now, the output of the forced turn alternation should start to look like Figure 5.8. The behaviour should be that you can only play a marker in an empty square, and the player turns alternative.

<sup>14</sup>Note that we need only have one occurrence of the line `player1 = ~player1;`, although it would have still worked fine to have put this line at the end of the code in the `if` section, and also the `else` section.

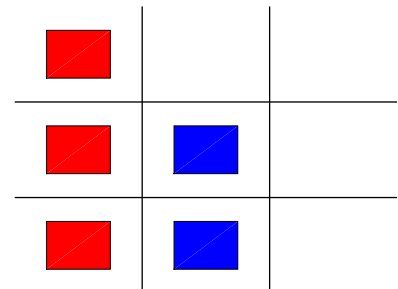


Figure 5.8: Tic-tac-toe game – object drawing now arranged in a grid and with forced alternation in player turn.

Finally, we need to think about the end-game. One way in which the game ends, is if there are no free cells left. We could:

1. Search through the  $3 \times 3$  grid, testing each cell in turn as to whether it has a value of zero or not.
2. `find` ... find the vector of indices of locations in which a value of zero occurs, and test whether this vector is empty<sup>15</sup>:

```
remaining = find(tokens == 0);
if isempty(remaining),
    break;
end
```

You could also add a message before `break`-ing out of the `while` loop.<sup>16</sup>

This code would go just before the end of the `while` loop and either just before, or just after the line:

```
player1 = ~player1;
```

---

Almost almost there ... and perhaps the single most hardest part – detecting a ‘win’.<sup>17</sup>

Probably, the easiest way, which would not be true for many other games, is to pre-define the every single winning pattern, and look through this list to see if any of these patterns occurs. For instance, one winning pattern is shown in Figure 5.8 and would be represented in matrix form by:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

We could define a series of  $3 \times 3$  arrays to represent these. We’d end up with 8 different arrays and array names and that might be a bit of a mess to deal with. Better is to create a single 3D array, with the 3rd dimension having length 8 – one layer for each possible winning pattern.<sup>18</sup> If you did this (create a  $3 \times 3 \times 8$  array ... we’ll call it array variable `winning`), we could simply loop through the 3rd dimension of the array, from 1 to 8 (in terms of the layer number), and access each possible solution in turn. How to use this in practice? Well, it is not obvious.

A slightly different alternative is to use `find`, and for each player, obtain the list of ‘linear indices’ (see below) of the grid cells containing their symbol.

For instance, for player 1 (red symbol) in Figure 5.8, we could create the matrix of the location of their marked squares and use it to explore strategies for determining whether it constitutes a ‘win’. The matrix would look like this:

<sup>15</sup> See Box. Note that if `find` finds nothing, it returns the empty vector `[]`.

<sup>16</sup> An alternative to the use of `break`, would be to set the value of `game_on` to false.

#### `isempty`

**MATLAB** says: ‘Determine whether array is empty’, and:

`TF = isempty(A)` returns logical 1 (true) if A is an empty array and logical 0 (false) otherwise.

<sup>17</sup> And don’t panic ... because the code will be given to you at the end because it is not at all simple (although you might yet come up with a much better solution ...).

<sup>18</sup> If you like – make an analogy with the month temperature data, where you had 12 slices (the 3rd dimension) of a 2D (*lon,lat*) array of points.



```
A = [1 0 0; 1 0 0; 1 0 0];
```

If you then do `find` for 1s (player 1's token value), you get:

```
> find(A==1)
ans =
     1
     2
     3
```

So for the winning 2D pattern in Figure 5.8, we could represent this more simply as `[1 2 3]`.

Be careful here – for 2D (or higher dimension) arrays, rather than the simple (1D) vectors you have been throwing at it previously – `find` returns the 'linear indices' of the array locations where the condition is true. For a *linear index* – rather than giving a (row, column) index, **MATLAB** counts continuously, down the rows in the first column, then down the next column, etc etc, to give an index as shown in Figure 5.9.

If we define a winning pattern as its 3 linear indices:

```
winning = [1 2 3];
```

(for the first pattern), we can search the game grid at the end of the each player turn and look for whether this (winning) pattern occurs anywhere. We will use the **MATLAB** function `ismember` like this:

```
> ismember(winning, find(A==1))
ans =
     1     1     1
```

where the three 1s indicate that each of the elements of `winning`, do indeed appear in the result of `find(A==1)`<sup>19</sup>. Only if three *true*s (1s) are returned, does the pattern completely match. To test for this condition, we can calculate the sum of the result of `ismember` and determine whether this is equal to 3. This then indicates to us that the winning pattern exists.

Your only job (as the code needed is given below) – is to define the  $8 \times 3$  array `winning`, which will contain all the 8 different winning patterns (rows), in terms of a linear index (i.e. what `find` returns) (see margin clues).<sup>20,21</sup> For instance, the 3 different winning column patterns would be represented by:

```
winning = [1 2 3 ; 4 5 6; 7 8 9];
```

and you need to add (in the same array, `winning`), the 3 winning row patterns (in linear indices), plus the 2 diagonals, for a total of 8 different patterns of 3 indices.

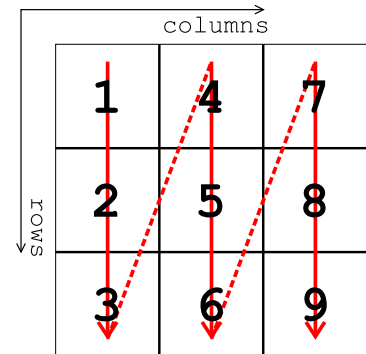


Figure 5.9: Linear indices of a  $3 \times 3$  matrix.

<sup>19</sup> The '1' because this is the notation for player 1.

<sup>20</sup> The way to go about it is to create a single winning pattern, and test the code and that it works, then define the remaining 7.

<sup>21</sup> Also – write down on paper, the linear indices of the  $3 \times 3$  array – that will help, e.g.:

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

198 str = 'do you like bananas?'

The code for player 1, which comes in the program (within the loop), just after you have set the value of the current cell in `tokens` to 1 (i.e. after player 1 has just taken their turn and you want to check whether they have just one):

```
pattern = find(tokens==1);
for n=1:8
    test_for_win = ismember(winning(n,:),pattern);
    if (sum(test_for_win) == 3),
        disp('Player 1 WINS!');
        game_on = false;
    end
end
```

(and similar code is needed for player #2 just after they have taken their turn). Note that if a winning combination is found, the game ends and `game_on` is set to `false`.

You could, of course, rather than write out this code twice, once for each player, create a *function*, passing in the player number, the current `tokens` array (`tokens`), and the winning patterns (`pattern`), and perhaps returning a *logical* value representing whether a win had occurred.

---

It is also left up to you, to improve the shapes/symbols used to mark the squares of each player. The cross is relatively easy. The circle is harder.<sup>22</sup>

<sup>22</sup> A polygon with a very large number of sides would do.

6

*Numerical modelling – zero-D / equilibrium*

## 6.1 Zero-D Energy-balance model of 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) (Figure 6.1). The equations are outlined in the Box in the margin, 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, and which are *functions* and which are *script* files ...

The sequence of work is as follows:

- 6.1.1 In this first Subsection ('*The basic EBM*'), you'll create a *script* (# `scr_1`<sup>1</sup>) **m-file** containing the Energy Balance Model (EBM), and test it.  
(See Figure 6.2.)
- 6.1.2 Next, you'll turn your EBM *script* (`scr_1`) into a *function* (`fun_1`)<sup>2</sup> – passing in the solar constant and albedo as parameters, and returning the surface temperature. (And test it.)  
(See Figure 6.3.)
- 6.1.3 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 6.4.)  
And then ...
- 6.1.4 ... 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 the surface temperature at time  $t$  –  $T_{(t)}$ .  
(See Figure 6.5.)

You can also take the EBM function (now ignoring the solar constant function), and play some theoretical games with it in order to understand how sensitive global surface temperature is to key variables (solar constant and albedo):

### Energy balance modelling (1)

The surface energy budget at the Earth's surface, to a zero-th order approximation, can be thought of as a simple balance between incoming, short-wave radiation that is \*absorbed\*, and out-going, infra-red radiation.

On average (over the Earth's surface and annually), the energy flux per unit area received from the sun, can be written:

$$F_{in} = \frac{(1-\alpha) \cdot S}{4}$$

where  $S$  is the solar 'constant' which has a present-day value (given the notation  $S_0$ ) of  $1368 Wm^{-2}$

(NOTE: 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 ( $\alpha$ ), is the fraction of incoming solar radiation that is reflected back to (-wards) space – 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  $\epsilon$  is the emissivity,  $\sigma$  is the Stefan-Boltzmann constant (in units of  $Wm^{-2}$ ), and  $T$  the temperature in Kelvin ( $K$ ) ( $273.15K == 0.0^\circ C$ ).

For a perfect black body radiator, we would set  $\epsilon=1.0$ . 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:

$$F_{out} = 0.62 \cdot \sigma \cdot T^4$$

See Figure 6.1.

<sup>1</sup> 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 ...

<sup>2</sup> **Once the EBM function has been created, you do not at any point edit it any further!**

6.1.5 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.<sup>3</sup> (See Figure 6.7.)

6.1.6 Then, in ‘Parameter sensitivity experiments using the EBM – #2’ – 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 be something of a diversion in this Subsection that will further help illustrate nested loops for you. (See Figure 6.9.)

### 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 6.2.<sup>4</sup> The equations for in-coming and out-going radiation (energy) were given previously. You simply need to re-arrange these in terms of  $T$  (i.e.  $T = \dots$ ) and write them as code. This will form the basis of subsequent, more complex (and later, time-stepping) models. In detail:

You are given:

$$F_{in} = \frac{(1-\alpha) \cdot S}{4}$$

and

$$F_{out} = \epsilon \cdot \sigma \cdot T^4$$

and are told at equilibrium:

$$F_{in} = F_{out}$$

You can then write:

$$\frac{(1-\alpha) \cdot S}{4} = \epsilon \cdot \sigma \cdot T^4$$

This equation now needs to be re-arranged in terms of  $T$ .

How to write the math down as **MATLAB** code? For the first part, we could e.g. write:

```
Fin = ((1-albedo)*solar_constant)/4;
```

and pretty well much as you would write as math. Here, `albedo` and `solar_constant` are variables holding the values of planetary albedo and solar constant, and the result of the calculation is assigned to a variable `Fin`. For completeness, you would define these values, e.g.

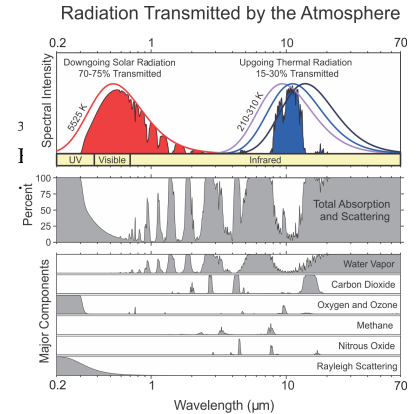


Figure 6.1: The pattern of absorption bands generated by various greenhouse gases and aerosols (lower panel) and how they impact both incoming solar radiation (upper left) and outgoing thermal radiation from the Earth's surface (upper right). (Figure prepared by Robert A. Rohde for the Global Warming Art project).

<sup>4</sup>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 = ...
```

202 str = 'do you like bananas?'

```
albedo = 0.3;  
solar_constant = 1368.0;  
Fin = ((1-albedo)*S)/4;
```

For writing out the  $F_{out}$  part of the equation in code, you will need to find (from the Internet?) the value of the (Stefan-Boltzmann) constant. When you have found this, assign this value<sup>5</sup> to a variable, e.g.

```
sb_constant = 9.9999E19;
```

... and you will need to be careful with units of this. Use a 'dimension' check to see if you have the units correct. In the equation:

$$F_{out} = \epsilon \cdot \sigma \cdot T^4$$

– on the left hand side we have units of  $Wm^{-2}$ , and on the right hand side  $\epsilon$ , the emissivity, is dimensionless, and  $T^4$  has units of ...  $K^4$ .

The Stefan-Boltzmann constant must be in units that balance this, i.e.  $Wm^{-2}K^{-4}$ .

Also note in the context of the Stefan-Boltzmann constant, how scientific notation (floating point) numbers are dealt with in **MATLAB**. For example, while in normals maths speak, you might write:

$$x = 9.9999 \times 10^{19}$$

in **MATLAB** you would write:

```
x = 9.9999E19
```

although, you could also write this out long-hand and more like maths speak if you prefer, e.g.

```
x = 9.9999*10^19
```

For now – prescribe the value of  $S$  (variable: solar\_constant) – for which the modern value is  $1368 Wm^{-2}$  ( $S_0$ ) as well as the value of surface albedo ( $\alpha = 0.3$ , variable: albedo) – somewhere near the start of the program (see Figure 6.2). 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  $14^\circ C$ <sup>6</sup> ... 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 source of errors is incorrect placement of parentheses ... or not placing parentheses around multiple variables you are divining something all by.

<sup>5</sup> Obviously, in this example, this is NOT the actual value ...

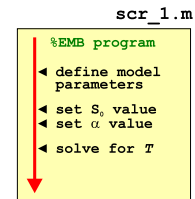


Figure 6.2: Form of the basic EBM model.

<sup>6</sup> Remembering to convert from Kelvin ( $K$ ) to degrees Centigrade ( $^\circ C$ ). In the equation you have re-arranged,  $T$  is in units of Kelvin ( $K$ ).

If it helps you to avoid confusion and potential errors and bugs by breaking down calculations into multiple steps using temporary/intermediate variables and partial calculations ... then do it!

3. If still 'no' – maybe take the 2 component equations (for  $F_{in}$  and  $F_{out}$ ), plug  $S$  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.

Once it is working, have a quick play about, changing the value of  $S$  and albedo ( $\alpha$ ) (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.

### 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*<sup>7</sup> that takes in 2 parameters – the solar constant ( $S$ ) and the mean global planetary albedo ( $\alpha$ ) (see hint in the margin!!). The *function* should return the global mean surface temperature,  $T$ .<sup>8</sup> (See Figure 6.3) Remember that you can directly replace the symbols in an equation with variable names in **MATLAB**, e.g.

```
S → solar_constant
α → albedo
T → temp
```

(or whatever you like, as long as the names help you in the coding and debugging). Or if you prefer – create a new (empty) *function* (select New and then Function from the **MATLAB** toolbar/menu-bar) and then copy-paste in the contents of `scr_1.m`. Note that in the *function*, you no longer define yourself the values for `solar_constant` and `albedo` in the **m-file** – instead, these values are passed into the *function* when you call it. For instance, if at the top of the *function* (see side-note) you defined:

```
function [temp] = fun_1(solar_constant, albedo)
```

then when you call the *function* at the command line:

```
» fun_1(1368.0,0.3)
```

you are passing in the value 1368 – assigned to the variable `solar_constant` (in `fun_1.m`), and the value 0.3 – assigned to the second variable in the *function* definition (`albedo`). The *function* then returns whatever value you have calculated and assigned to the variable `temp`, back to you (and which then appears at the command line).

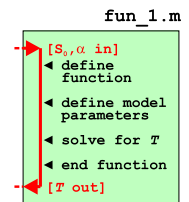


Figure 6.3: Form of the basic EBM model as a *function*.

<sup>7</sup> 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)
```

<sup>8</sup> 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).

Try playing with the *function* in the same way as before, but now passing the different values of  $S$  and  $\alpha$  (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:  $S$ ,  $\alpha$  and are given their default values, you'd write (at the command line):

```
» temp = fun_1(1368.0,0.3)
```

(and get a value close to 14°C returned, and if not – debug it ...).

### 6.1.3 Creating a function for the evolution of solar constant through geological time

In this sub-subsection, as a precursor to simulating how Earth's surface temperature may have changed through geological time, you are going to code up a new *function* that calculates (and returns) the value of the solar constant as a function of 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 a variable for time,  $t$  (in units of *Ga*) and spits out (aka, returns) the calculated value of  $S_{(t)}$  ( $Wm^{-2}$ ) (this *function* will be 'fun\_2' in the on-going notation and obviously saved as `fun_2.m`). i.e., your *function* definition (at the top of the **m-file**) will look something like:

```
function [St] = fun_2(t)
```

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 ( $S_{(t=0)}$  or  $S_0$ ) into the equation to leave it in terms of  $S_{(t)}$  (the solar constant value at time  $t$ ).

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  $S_{(t=0)}$  and perhaps the reference time<sup>9</sup> ( $t_0$ ) (4.57 Ga) ... and a single line for the equation giving the value of  $S_{(t)}$ :

$$S_{(t)} = \frac{S_0}{1 + \frac{2}{5} \cdot (1 - \frac{t}{t_0})}$$

As before – convert this into **MATLAB** code. You could either create parameters (variables) containing the values of  $S_0$  and  $t_0$ , or plug them directly into the code. For the former, translating the equation into **MATLAB** might look like this:

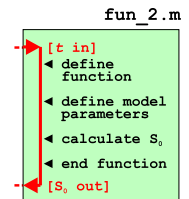


Figure 6.4: Schematic structure of code for calculating the solar constant (output) as a function of time (input).

#### 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 ( $4.57 \times 10^9$  yr) and  $L_0$  is the present-day solar luminosity ( $3.85 \times 10^{26}$  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 – the present-day solar 'constant'  $S_0$ . In the equation,  $L$  can hence be substituted for  $S$  to give the value of  $S$  ( $Wm^{-2}$ ) at any time ( $S_{(t)}$ ), i.e.

$$\frac{S_{(t)}}{S_{(t=0)}} = \frac{1}{1 + \frac{2}{5} \cdot (1 - \frac{t}{t_0})}$$

or, in terms of the value of  $S$  at time  $t$  and using using the notation  $S_0$  in place of  $S_{(t=0)}$ :

$$S_{(t)} = \frac{S_0}{1 + \frac{2}{5} \cdot (1 - \frac{t}{t_0})}$$

Note that in the formula,  $t$  is counted (in Gyr) relative to the formation of the Sun (i.e. present-day would be:  $t = 4.57$ ).

The reference value of  $t$ :  $t_0$ , is  $t_0 = 4.57$  Gyr.

The reference value of  $S$ :  $S_0 = 1368 Wm^{-2}$ .

<sup>9</sup> Which is also equal to the current time (since the formation of the Sun).



```

ref_S0 = 1368.0;
ref_t = 4.57;
St = ref_S0 / ( 1 + (2/5)*(1-(t/ref_t)) )

```

Here, the value of  $t$  used in the equation is passed in as per defined in the *function* header (see above). The result of the equation – the value of the solar constant at that time, is assigned to the variable  $St$  and passed out as the result of the function (again, as per the function definition/header).

In the code and in general – use as many pairs of (nested) parentheses as you need to help make the equation clear. You can also use spaces to help make it clearer which end parenthesis corresponds to which opening parenthesis

When you think you have done this – check it – plug in values of time into your function, i.e.

```
» St = fun_2(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 ( $Wm^{-2}$ )).

As a little test – see if you can adjust your function so that rather than passing in time, measured since the formation of the Sun, you pass in time relative to now (i.e. » fun\_2(0.0) would then give you a value of 1368). Hint: the time used in the equation, must be as Gyr following the formation of the sun, but the value you are passing in, has had a value of 4.57 (Ga) removed from it to make it relative to now. Hence, before the calculation in the function, you need to add this value back to the time variable passed in, to make it absolute rather than relative time again.

#### 6.1.4 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 then at the end (of the program) plot, the surface temperature of the Earth, at 100 Myr intervals, and spanning approximately the age of the Earth and much of its potential long-term future.

Start by creating a new (yet another blank **m-file**) *script* ('scr\_4')<sup>10</sup>. You are going to need a loop in time (e.g. with the variable name  $t$ ), perhaps looping from 0.0 (the age of formation of the Sun) to 10.0 Ga (with the step size being 0.1 Ga). Within the time loop, you will:

1. Pass to your solar constant *function* (fun\_2.m) your variable containing the current value of time ( $t$ ), and obtain the corresponding value of the solar constant ( $S_{(t)}$ ), and assign to a variable e.g.  $St$ .

<sup>10</sup> The structure of the overall program is shown in Figure 6.5.

Note that you do not copy-paste the `fun_2.m` code into `scr_4.m` ... you simply call the *function* within the loop exactly as per you did at the command line (but assign the result to a variable), e.g.

```
St = fun_2(t);
```

(NOTE: The summary figure (Figure 6.5) is intended to indicate the flow of information (variable values) and relationship between the *script* and two *functions* **m-files**, rather than that the code for the 2 functions should be embedded (which it should not) within the actual *script* file itself ...)

2. Call your zeroD EBM *function* (`fun_1.m`) to calculate the corresponding surface temperature, passing it the value of  $S_{(t)}$  that you have just calculated and assigning the result to e.g. `temp`:

```
temp = fun_1(St,albedo);
```

(or simply replace `albedo` with a fixed value, e.g. 0.3).

3. Store in an array, or pairs of vectors, the current time in the loop alongside the corresponding value of  $T$ . For hints on the various different possibilities in doing this see earlier in the text, but you might e.g. do something like:

```
vt = [vt t];
vtemp = [vtemp temp];
```

(having first (before the loop) initialized these vectors as empty, e.g. `vt=[]`; and `vtemp=[]`).

Once the loop has completed, plot surface temperature ( $y$ -axis) as a function of time ( $x$ -axis).

Likely bug possibilities include mistakes with nested parentheses (`()`), units (e.g.  $K$  vs.  $^{\circ}C$ ), and time, which for now should run forward from zero (the formation of the Sun). A schematic of the program structure is shown in Figure 6.5 to aid you.

Assuming that you have managed something like Figure 6.6<sup>11</sup> – 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’?

As an additional step and noting that the time-scale is not entirely helpful in terms of knowing when ‘now’ is, you could:

1. Draw on a vertical line (hold on) at 4.57 (‘now’, relative to the time of formation of the Sun).
2. Transform the  $x$ -axis time scale to time relative to now. To do this – as you loop through time relative to the formation of the Sun, when you save the current time for plotting, you could subtract 4.57 from the loop value before passing it to `fun_2.m`.

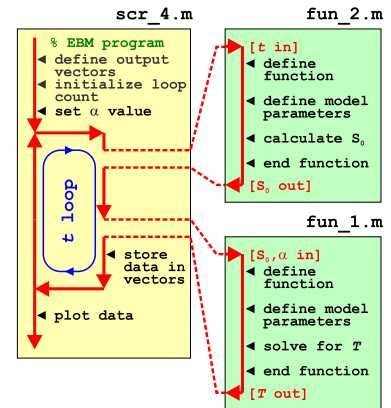


Figure 6.5: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, and solar constant and EBM functions. Note – in this schematic, the code contents of the two *functions* remain in their respective **m-files**. The *function* code does not get copy-pasted into the `scr_4.m` scripting file. The red arrows indicate the passing of variable values ... from `scr_4.m` into each *function*, with the *functions* returning variable values back to `scr_4.m`.

<sup>11</sup> Note that a line has been added to highlight  $t = 0$  (i.e. the present-day) – see line (see earlier). This plot also has an altered time-axis and time is plotted relative to now – see below.

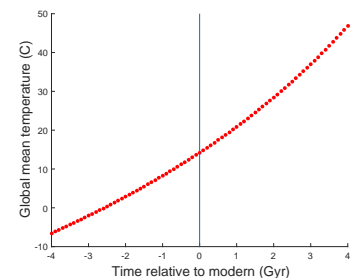


Figure 6.6: Simple EBM projection of the evolution of Earth surface temperature with time. Time at the present-day is highlighted by a vertical line (drawn using the **MATLAB** `line` function).

3. Or ... you could save the time as given in the loop, but transform the  $x$ -axis time scale to time relative to now by subtracting a value of 4.57 when you come to `plot` it, e.g.:

```
» plot(x-4.57,y);
```

or more explicitly so you can see what is going on:

```
» plot((x(:)-4.57),y(:));
```

Note that you do not have to plot the entire dataset and could set the  $x$ -axis limits to e.g.  $-4 \rightarrow +4$  Gyr relative to present.

208 str = 'do you like bananas?'

### 6.1.5 Parameter sensitivity experiments using the EBM – #1

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  $S$  on its own, or changing  $\alpha$  on its own, but what about when both parameters vary together? In this exercise you are going to utilize your energy-balance model *function* ('fun\_1' in the summary notation) to explore this.

Create a new blank script ('scr\_2') and define 2 parameters near the start of the **m-file** – one for the value of  $S$  and one for the albedo,  $\alpha$ , then further down the code, call your function (fun\_1), passing it these 2 parameters but remembering that you need to assign the result of your function (fun\_1) to some *variable*<sup>12</sup>. So far so boring, as this is in effect what you had been doing previously in 'playing' with the energy balance function.

Starting with a simple 1-D case and considering just a progressive change in the value of  $S$ , to automatically generate a progression of different values of  $S$  and call the *function*, you are going to need to create a loop<sup>13</sup>. There are two/three ways of constructing the loop<sup>14</sup>:

**loop option #1** You could loop directly through the range of values of  $S$  that you are interested in, e.g.

```
for S = 1000:100:1500
    % CODE GOES HERE
end
```

in which case,  $S$  will go from 1000 to 1500 ( $Wm^{-2}$ ) in steps of 100 ( $Wm^{-2}$ )<sup>15</sup>.

Perhaps a little inconveniently, this does not pass exactly through the modern value ( $1368 Wm^{-2}$ ), although when you plot as a continuous line (e.g. in plot), maybe this does not matter. Remember, you could also interpolate the result later (e.g. on a new vector of solar constant values that include 1368).

You could have addressed this by constructing a slightly less convenient form of the loop, e.g.:

```
for S = 1068:100:1568
    % CODE GOES HERE
end
```

which now passes exactly through the modern value of  $S$ .

Or ... you could have made the loop go around in steps of 1 (and hence passing through a value of 1368) for a total of 501 loop iterations(!) But this is over-kill in terms of data generation if the calculated equilibrium is not particularly sensitive to the value of solar constant (i.e. not highly non-linear).

<sup>12</sup> i.e.

```
temp = fun_1(S,albedo);
```

assigns the result of your temperature calculation to the variable temp.

<sup>13</sup> You are going to put the loop in the script (# scr\_2), NOT the function (# fun\_1).

**An entire plane of Hell is reserved for anyone coding the loop in the function.**

<sup>14</sup> In both cases a for ... loop.

<sup>15</sup> You can pick a different range and increment ... this is just a quasi-random example to illustrate ...

**loop option #2** Alternatively, you could have an integer count for the loop, and then derive a value of  $S_0$  from this. For example:

```
S_modern = 1368.0;
for m=-5:5
    S = S_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  $S$  ( $S_0$ )<sup>16</sup>, meaning that  $S$  ends up going from  $S_0_{\text{modern}} - 500$ , to  $S_0_{\text{modern}} + 500 \text{ Wm}^{-2}$  (in steps of  $100 \text{ Wm}^{-2}$ ).

<sup>16</sup>The variable definition  $S_0_{\text{modern}} = 1368.0$  at the top of the code fragment.

**loop option #3** Or ... as a variant on #2:

```
S_modern = 1368.0;
for m=1:11
    S = S_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  $1$ .

To practice your coding skills – try coding up all 3 variants and satisfy yourself that you are happy how they all work, and how they are all equivalent to each other.

So what does it matter, and/or is one 'better' than the others? Although the all are in effect equivalent, the advantage with the second (and third versions) is that you explicitly have an integer counter. For the first version, you'd have to add lines, e.g.:

```
count = 0;
for S = 1068:100:1568
    count = count + 1;
    % CODE GOES HERE
end
```

in order to have a loop count.

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  $S$ . This data will need to go into an array, with one row corresponding to each value of  $S$ .

As per constructing the loop itself, there are also multiple (two-and-a-bit) obvious alternative ways of saving the data (and assigning calculated values to sequential locations in an array):

**save option #1** In this, you create the necessary array(s) beforehand, e.g. using the *zeros function*. For instance, to create a vector with

```
210 str = 'do you like bananas?'
```

11 rows (and 1 column), suitable for saving the value of  $T$  calculated by each call to your EBM function (`fun_1.m`), you could write:

```
data_T = zeros(11,1);
```

which would create a (single) column vector with 11 rows. You'd also need an equivalent vector (e.g. `data_S` in this example) for storing the corresponding value of  $S$  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  $S$  and  $T$  by using whichever index you created<sup>17</sup>:

```
data_S(m) = S;  
data_T(m) = temp;
```

or:

```
data_S(count) = S;  
data_T(count) = temp;
```

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** Related to the above – you should recognise that creating 2 separate vectors is messy, when you could easily create just a single matrix instead. To create the blank array, we would now write:

```
data = zeros(11,2);
```

which creates a matrix of zeros of 11 rows by 2 columns.

Within the loop, `data` is now assigned:

```
data(m,1) = S;  
data(m,2) = temp;
```

(where the first column is used to store the solar constant value, and the second the corresponding temperature value).

**save option #3** Or ... **MATLAB** will allow you to 'grow' a vector, one element at a time (but not for matrices).<sup>18</sup> The code within the loop actually looks identical, but instead of creating a pair of vectors (or a matrix) of a size (number of rows) that matches the number of iterations of the loop, you create an empty vector (or matrix)<sup>19</sup>:

```
data_S = [];  
data_T = [];
```

and then within the loop:

```
data_S = [data_S; S];  
data_T = [data_T; temp];
```

<sup>17</sup> i.e. which of the loop OPTIONS you chose earlier.

<sup>18</sup> 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 of 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` ...

<sup>19</sup> Try the code without creating empty vectors at the start, and see what happens? Why is **MATLAB** unhappy?

Note that you cannot grow a matrix by adding data for a single cell, as a matrix always has to have a complete number of rows and columns. Instead, you'd have to write:

```
data = [];
```

during initialization before the loop starts, and then with the loop:

```
data = [data; S temp];
```

i.e. concatenating a vector `[S temp]` (and hence a complete row) to the end of the matrix `data`.

Pick one of these (i.e. a way of saving a pair of values each time around the loop) and code it up (or better, try all of them in turn!).

Finally, at the end of your program (after the end of the loop), you can now plot (`plot` or `scatter`) how  $T$  varies as a function of  $S_0$ , having saved all the values of  $S$  you tested, plus the corresponding calculated temperatures, in a handy matrix (or pair of vectors). Note that regardless of whether you use `plot` or `scatter`, you need to specify to **MATLAB** that you want the values in the 2nd column of the array ( $y$ -axis), plotted against the first column ( $x$ -axis).<sup>20</sup>

The structure of your code should look like Figure 6.7. and your resulting figure (depending on the range you assume for  $S$ ), something like Figure 6.8.

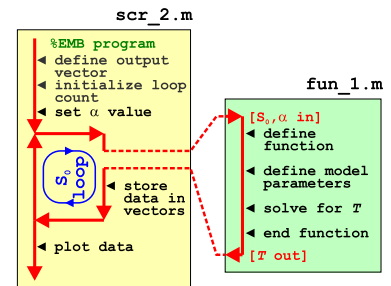


Figure 6.7: Schematic structure of the model configured to carry out a single parameter sensitivity study.

<sup>20</sup> Refer back to the earlier chapters on plotting of e.g. the CO<sub>2</sub> or global temperature data you analysed, and also recall how to specify e.g. all the rows in the first column of an array.

### 6.1.6 Parameter sensitivity experiments using the EBM – #2

In this final sub-subsection, we'll extend the parameter sensitivity analysis of your model to 2D, assuming for instance that you are now interested in how  $T$  also varies both as a function of solar constant as well as, as a function of  $\alpha$  (surface albedo). You'll need to vary both  $S$  and  $\alpha$ , and in all combinations of the two in order to achieve this. In fact, you'll do this in a grid pattern, with  $S$  increasing in steps on one axis (as before), and  $\alpha$  on the other.

Hopefully, you might have guessed that you'll need a *nested loop*(?) – one loop going through all possible values of  $\alpha$ , for each and every possible value of  $S$ ?? i.e. in a structure like:

```
for ...
  for ...
    % CODE GOES HERE
  end
end
```

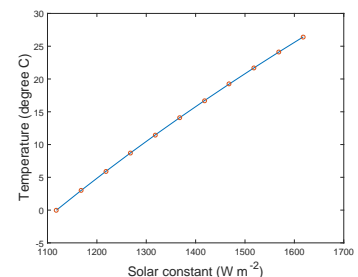


Figure 6.8: Sensitivity of global mean surface temperature vs. solar constant (mean surface albedo held constant at an albedo value of 0.3).

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. Chose whether to use counters (e.g.  $n$  and  $m$ ) in the `for` loops and then derive the values of  $S$  and  $\alpha$  from these counters, or loop through the values of  $S$  and  $\alpha$  directly and create counters (as per for the 1D case). Call your *function* (`fun_1`) for solving the global surface temperature within the innermost loop (passing it the values of  $S$  and  $\alpha$  generated in the loop). A schematic of the program structure is shown in Figure 6.9.

For saving the data (within the loop), you cannot simply index the locations you want in a 2D array (matrix) that did not previously exist and expect it to 'grow' as before, because a matrix must have all complete rows and columns and you are generating the results (value of  $T$ ), one cell at a time, while you'd need a complete row or column of results in order to append to the results array. 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  $S$  and 10 different values of  $\alpha$ , you could write:

```
data_output = zeros(10);
```

(creating a  $10 \times 10$  array of zeros). Or if for example, you had 20 different values of  $S$ , and 10 of  $\alpha$ :

```
data_output = zeros(10,20);
```

(20 columns times 10 rows).

Within an (e.g.  $n, m$  loop if you did it that way), you then assign your calculated value of  $T$  to the appropriate location in the array:

```
data_output(n,m) = temp;
```

Don't forget that you'll also need to know the values of  $S$  and  $\alpha$  that correspond to the column and row numbers. Perhaps save these as 2 individual vector (as per before), or create 2D arrays for them with each element corresponding to an element in the `data_output` array, or simply just ignore them for now.

One slight complication if you chose to employ a pair of counters for indexing the results array, and increment their value each time around their respective loops (rather than having a integer count for the loop itself (i.e.  $n$  and  $m$ ) and derive the actual values of  $S$  and  $\alpha$  from that) – the innermost counter must be reset in value each time the outer loops starts. This would look like:

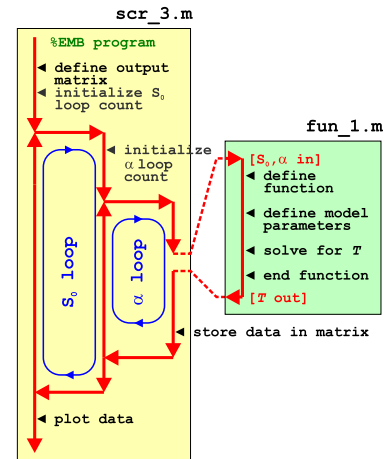


Figure 6.9: Schematic structure of the model configured to carry out a double (in terms of solar constant AND now albedo) parameter sensitivity study.



```

count_outer = 0;
for ...
    count_outer = count_outer + 1;
    count_inner = 0;
    for ...
        count_inner = count_inner + 1;
        % CODE GOES HERE
    end
end

```

Be careful here that you increment the value of the count variable before using it to index the position in an array – an index of zero is invalid in **MATLAB**. Or, you could initialize the count variable to a value of 1 before the start of the loop and increment its value after you use its value to index a location in the results array.

When you *\*think\** you have this working and have generated a matrix of  $T$  values<sup>21</sup>, plot the resulting surface of  $T$  vs. the two parameters. Rather than using e.g. `imagesc` (Figure 6.11)<sup>22</sup>, try `contour`<sup>23</sup> or `contourf` (e.g. Figure 6.10).

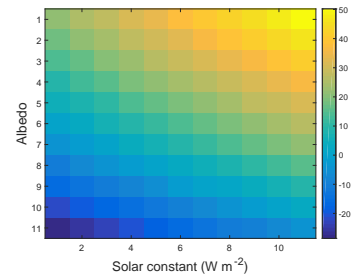


Figure 6.10: Global mean surface temperature ( $^{\circ}\text{C}$ ) as a function of solar constant and surface albedo grid point number.

<sup>21</sup> 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.

<sup>22</sup> 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 `contourf` version.

<sup>23</sup> You'll need to employ `meshgrid` based on the same 2 vectors of values that the loop creates for  $S_0$  and  $\alpha$ .

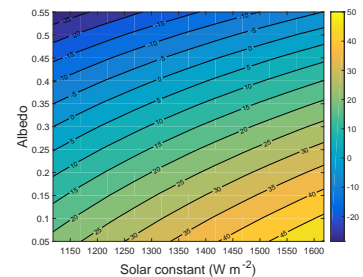


Figure 6.11: Global mean surface temperature ( $^{\circ}\text{C}$ ) as a function of the value of solar constant and surface albedo.

## 6.2 'Daisy World'

There is an absolutely classic paper from the early 1980s – *Watson et al.* [1983] – that illustrates how simple (biological) feedback on the climate system can lead to a close regulation of global climate over an appreciable span of the Earth's 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) as per Figure 6.12<sup>24</sup>. 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).<sup>25</sup> Note that the *script* and *function* naming convention continues on from the previous section and the zero-D energy balance model. (You will, of course, have had to have completed up to an including Section 6.1.4.)



Figure 6.12: Daisy World

<sup>24</sup> As pointed out in *Watson and Lovelock* [1983], the actual 'colors' are immaterial – just that their albedos differ.

<sup>25</sup> Note that what immediately follows is just a summary list ... not the instructions themselves ...

<sup>26</sup> 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 albedo) and solve for mean global surface temperature. This was '# scr\_4' in the previous Section notation.

6.2.1 This will be the simplest addition to your previous model<sup>26</sup>. You'll create a new 'fixed daisy' function (here called `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 your previous m-file `scr_4`) and in it, take the albedo value generated by the call to the daisy function, and pass it into your EBM function (m-file `fun_1`).

(See Figure 6.13.)

6.2.2 Now, in the next stage it gets a little more complicated, because in a further new function ('`fun_4`' – copy-renamed-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 and incorporates some population dynamics of the daisies.

In the main (time since the Sun formed) loop (in `scr_5`), the situation thus becomes – the relative fractions of dark and light colored daisies is now 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 in `scr_5` that comprises only the daisy (albedo) function (`fun_4`) and the EBM function and keeps looping until ... well, we'll start by simply prescribing a fixed number of iterations of the loop.

(See Figure 6.15 for a schematic of the code setup.)

6.2.3 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<sup>27</sup>, 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.

6.2.4 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).

OK then – here goes ...

### 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.

As summarized above – create a new function (`fun_3`) and configure it so that it returns a single parameter – albedo. For now, it has no inputs.<sup>28</sup> How it relates to your previous program and code for how the Earth's surface temperature evolves over geological time, is illustrated in Figure 6.13.

In the daisy/albedo 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_b = 0.1; % albedo - black daisies
par_a_w = 0.5; % albedo - white daisies
% define model parameters - daisy land fraction
Fb = 0.01; % (land) fraction - black daisies
Fw = 0.01; % (land) fraction - white daisies
```

<sup>27</sup> 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.

<sup>28</sup> A funny sort of function, although pretty well much like pi.

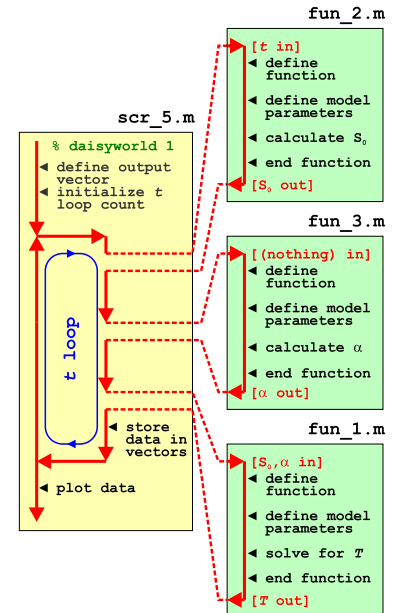


Figure 6.13: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, the solar constant and EBM functions, and now the 'daisy' albedo function.

(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 give 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.<sup>29</sup>

These parameters relate to the symbols in the equations that follow, as follows:

$\alpha_s$  – par\_a\_s (albedo – bare soil)  
 $\alpha_b$  – par\_a\_b (albedo – black daisies)  
 $\alpha_w$  – par\_a\_w (albedo – white daisies)  
 $F_b$  – Fb (land) fraction – black daisies  
 $F_w$  – Fw (land) fraction – white daisies

Next, and actually the only line of any note in the *function* – you need to calculate the average albedo<sup>30</sup> – calculated based on the area 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 Earth's surface that is bare soil – the area fraction not covered by daisies. In maths-speak, the mean albedo is given by:

$$\alpha = F_w \cdot \alpha_w + F_b \cdot \alpha_b + (1.0 - F_w - F_b) \cdot \alpha_s$$

where  $\alpha_w$ ,  $\alpha_b$ , and  $\alpha_s$ ,  $F_w$ , and  $F_b$  are as defined above. Bare soil is simply whatever the fraction of the planet is not covered by daisies, i.e.  $(1.0 - F_w - F_b)$ .

You simply need to translate all this into **MATLAB** code using the parameters you defined earlier (for  $\alpha_w$ ,  $\alpha_b$ , and  $\alpha_s$ , and  $F_w$  and  $F_b$ ). The code will look pretty well much like the equation, but you substituting whatever variable/parameter names you have chosen for the symbols in the maths:

```
% calculate mean albedo
albedo = Fw*par_a_w + Fb*par_a_b + (1.0 - Fw - Fb)*par_a_s;
```

To be neater, we could also pre-calculate the fraction of bare ground,  $F_g$ , and make ourselves a slightly shorter (and easier-to-debug) mean albedo calculation, e.g.

```
% calculate fractional area of bare ground
Fg = (1.0 - Fw - Fb);
% calculate mean albedo
albedo = Fw*par_a_w + Fb*par_a_b + Fg*par_a_s;
```

<sup>29</sup> 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!).

<sup>30</sup> Note that it is very easy to accidentally 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`).

Add these lines of code (immediately after the 5 lines of parameter definitions), which will be the one and only calculation that this particular **MATLAB** function ((fun\_3), Figure 6.13) carries out, just before the `end` of the function.

That is actually it. All the variable/parameter values are specified and fixed in the daisy function (see above), so nothing particularly exciting is going to happen ... Regardless – run the complete model (i.e. » `scr_5`) which now has 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 6.14).

You might play briefly with the prescribed daisy area fractions ( $F_b$  and  $F_w$ ) and albedo values (`par_a_b` and `par_a_w`) 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

STEP #2 in the evolution of the Daisy World model, and for a modification 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  $S_0$ ). The daisy population is now going to grow and die (but unlike Southern California, not burn), with their relative fractions 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 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 (rather than an absolute number of individuals as we had before, but these are pretty much completely interchangeable). So from generation-to-generation (or on each subsequent time step, if you prefer to think of it that way), the fractional area of 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.<sup>31</sup> Similarly,

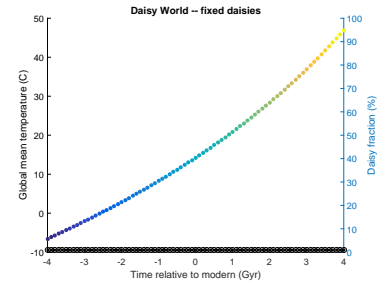


Figure 6.14: Evolution of global surface temperature and the two populations of daisies with time ... but with no change allowed in the daisy populations (d'uh!). The fractional coverage of white daisies is shown by large empty circles, and for black, by small filled black circles. Data points for mean 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:

$$\begin{aligned} dF_w/dt &= F_w \cdot (x \cdot \beta_w - \gamma) \\ dF_b/dt &= F_b \cdot (x \cdot \beta_b - \gamma) \end{aligned}$$

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),  $\beta$  is a temperature-dependent growth function (one for each species of daisy), and  $\gamma$  the mortality rate (as a proportion of the area covered by that species of daisy per unit time). The value of  $\gamma$  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):

$$\begin{aligned} \beta_w &= 1.0 - 0.003265 \cdot (22.5 - T)^2 \\ \beta_b &= 1.0 - 0.003265 \cdot (22.5 - T)^2 \end{aligned}$$

(where the value of 22.5 °C is a reference temperature and represents where optimal (maximum) growth occurs).

<sup>31</sup> Note the parallels with before – the new fractional area is dependent on the previous area, whereas before it was the new population size (number of individuals) that was dependent on the previous population size.)

mortality also depends on the current areas of daisies. The growth rate is further modified by the available fractional area, such that as 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 then to implement changing areas and growth of daisies in code? (We'll come to how to translate the equations into code after ensuring we have the basic structure of the program built. A general programming Plan of Action is given in the margin.)

Figure 6.15 gives a schematic of the overall code structure for this model. The new difficulty here is that the relative 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 resolve this circularity (i.e. come to a steady state solution) by creating an inner (nested) loop that comprises only the daisy function and EBM function.

**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 calculating the solar constant as a function of time (`f_un_2`) and the EBM model (`f_un_1`) (temperature as a function of solar constant and albedo) are exactly the same as before. The loop in (geologic time) and hence some of the script (`scr_6`) is also the same (as `scr_5`). What is different and yet to-do?

1. Lets start with the daisy (albedo) function (which will be based on the previous, `f_un_3` one). You could deal with the inputs and outputs first. As as well as  $T$ , the previous values of the fractional areas of the two daisies ( $F_w, F_b$ ) are also required by the function (which is different from before where the values were assumed and the respective parameters set at the start of the function<sup>32</sup>). This is because each time the daisy fractional area function is called, the fractional areas are updated (hence why they are inputs). And outputs. Because the daisy function is updating the fractional areas, these two parameters also need to be outputs too. So the very first thing to do is to modify the function definition, re-saving it as `f_un_4` (see Figure 6.15), so that the inputs are:

$$T, F_w, F_b$$

and the outputs are:

$$\alpha, F_w, F_b$$

Programming strategy:

- In general – start by identifying any constants – i.e. fixed and invariant, fundamental values, such as  $\pi$  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 to 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.
- 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.

<sup>32</sup> 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;
```



(see help of various sorts on *functions*, but it not at all a fundamental change as to compared to before). Of course, you need to substitute the maths symbols for the actual variable and parameter names you choose to use. So ... e.g.

```
function [albedo, Fw, Fb] = fun_4(temp, fw, fb)
```

which defines a *function* with 3 inputs, and 3 outputs.

Then, the only other development in the function, is to implement the equations for daisy growth/death (see Box) and update the values of  $F_w, F_b$ .

2. How to translate the given daisy population/growth equations into code? We could start by substituting the value of  $\gamma$  for its literature value of 0.3 to make it a little less scary. And also set the growth rate function,  $\beta$  to 1.0 for now, so that does not distract us either. The now simpler equations look like:

$$\begin{aligned} dF_w/dt &= F_w \cdot (x - 0.3) \\ dF_b/dt &= F_b \cdot (x - 0.3) \end{aligned}$$

which says that the change in fractional area ( $dF$ ), from one iteration (generation or time step) to the next is proportional to the current fractional area ( $F$ ) multiplied by some stuff ( $x - 0.3$ ).

We could re-write this in terms of a (loop) iteration number ( $n$ ) and also ignoring for now which daisy (black or white) we are talking about:

$$F_{(n+1)} = F_{(n)} + F_{(n)} \cdot (x - 0.3)$$

or rearranging:

$$F_{(n+1)} = (1.0 + x - 0.3) \cdot F_{(n)}$$

which says quite simply that the next fractional area estimate, is equal to the current one, multiplied by  $(1.0 + x - 0.3)$ . In code, this looks like:

```
for n=1:100
    F = (1.0 + x - 0.3)*F;
end
```

taking 100 loop iterations as an example. But ... we are not writing the population and albedo update code directly in the loop, but rather, it is going into `fun_4`, with that function called from within the `for n=1:100 ...` loop (Figure 6.15).

In `scr_6`, schematically, we have:

```
for t= ...
    St = fun_2(t)
    for n=1:100
        [albedo, Fw, Fb] = fun_4(temp, Fw, Fb)
        temp = fun_1(St, albedo)
    end
end
```

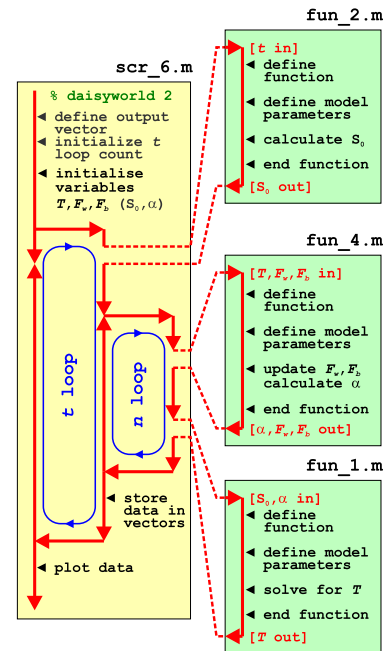


Figure 6.15: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, the solar constant, EBM, and ‘daisy’ albedo functions. Note the creation of an inner loop, with EBM, and ‘daisy’ albedo functions called from within this, while the solar constant remains called from the start of the outer loop as before.

which says that for every value of time (the  $t$  loop), you calculate the solar constant ( $S_t$ ) using your solar constant function. Then you are going to iterate (100 times here), updating the albedo with the initial temperature value, then recalculating the temperature using the new albedo value, then recalculating the albedo, then recalculating the temperature, etc etc ...

Within the function `fun_4` you should have:

```
F = (1.0 + Fg - 0.3)*F;
```

here substituting  $x$  for our previous variable name for the fraction of bare ground (the fraction of the planet not covered in daisies),  $F_g$ . And if we also bring both daisies and their respective fractional areas back into the picture:

```
Fb = 1.0 - Fb - Fw;
Fb = (1.0 + Fg - 0.3)*Fb;
Fw = (1.0 + Fg - 0.3)*Fw;
```

3. Now you are in a position to worry about the temperature dependent functions for growth, which were:

$$\beta_w = 1.0 - 0.003265 \cdot (22.5 - T)^2$$

$$\beta_b = 1.0 - 0.003265 \cdot (22.5 - T)^2$$

These are actually pretty simple – you take temperature, subtract it from a value of 22.5 and square it, multiply it by 0.003265 and subtract from 1.0 ...

```
bb = 1.0 - 0.003265*(22.5-temp)^2
bw = 1.0 - 0.003265*(22.5-temp)^2
```

Really – just as it looks written down mathematically. So now the content of `fun_4` will contain:

```
Fg = 1.0 - Fb - Fw;
bb = 1.0 - 0.003265*(22.5-temp)^2
bw = 1.0 - 0.003265*(22.5-temp)^2
Fb = (1.0 + Fg*bb - 0.3)*Fb;
Fw = (1.0 + Fg*bw - 0.3)*Fw;
```

4. So far, in `fun_4` you have updated the area fraction remaining (bare ground), updated the growth factors for the two species of daisy, and then updated the fractional areas of both species of daisy. Remaining, in this function, is to take the new fractional areas, and update the mean albedo (which is then returned from the function as an output):

```
% update mean albedo
albedo = Fg*par_a_s + Fw*par_a_w + Fw*par_a_b;
```

After this function returns the new updated values of mean albedo (and the two fractional daisy areas in case we want them for plotting later), the EBM function (`fun_1`) is called (in the inner loop) (Figure 6.15).



5. 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 value to be assigned.<sup>33</sup>

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 6.16.

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  $\alpha$  still never changes. One thing you could try to liven things up a little is to change on of the value of  $\beta$  (and/or  $\gamma$ ) 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.

### 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 6.17.

In the code – first copy `fun_4` → `fun_5`, and `scr_6` → `scr_7`, remembering to now call `fun_5` from within the inner loop in `scr_7`. (Otherwise, the structure of the model is the same as before.)

In `fun_5`, modify the equations of the growth factor  $\beta$  for each species of daisy as per the equations in the Box. Now, instead of using  $T$  (the global mean temperature) in both growth equations, each equation has its own local temperature – one associated with black daisies ( $T_b$ ) and one with white ( $T_w$ ). The local temperatures are calculated as deviations from the global mean, as per the equations in

<sup>33</sup> For completeness, you could also initialize  $S_0$  and  $\alpha$ , but it is not strictly needed, as they are calculated and defined before they are first used.

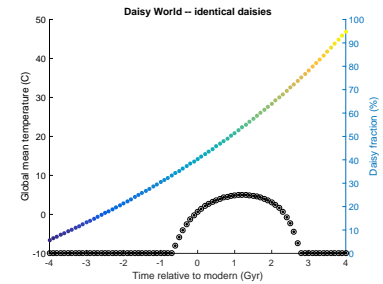


Figure 6.16: Evolution of global surface temperature and the two populations of daisies with time ... but now assuming that the growth of each depends on the global mean surface temperature.

#### 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 ways of defining how the local temperature deviates from 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 mean planetary albedo here, not  $\alpha$  as was the case in the original (non daisy enabled) EBM, while  $A_b$  and  $A_w$  are the albedos of black and white daisies, respectively).

$q$  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 different daisy fractions) and is assigned a default value of 10.0.

the Box. You'll need to calculate  $T_b$  and  $T_w$  in the code first, before calculating the values of  $\beta$ .

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 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  $22.5^\circ\text{C}$  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  $\beta_w$  and  $\beta_b$  you currently have values of  $22.5^\circ\text{C}$  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 6.17.)

#### 6.2.4 Efficient and 'clever daisy' daisy-world

The purpose of the inner loop is to calculate the equilibrium planetary temperature for each value of  $S_0$ . It may be that an equilibrium is reached much sooner than the 100 loop iterations that are allowed.

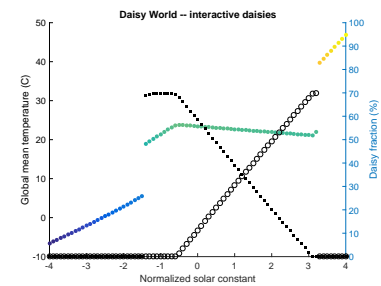


Figure 6.17: Evolution of global surface temperature and the two populations of daisies with time.

So rather than running the inner loop for the fixed number of iterations each time, you could make the overall calculation more efficient by testing whether the change in global temperature between one iteration and the next, is lower than some small threshold value – indicating that the iterative calculation has converged.<sup>34</sup>

---

NOTE that while the Daisy World equations can be written in terms of the population (or area fraction) at the  $n$ th generation, strictly, they are formulated in terms of the population (area fraction) at time  $t$ .

<sup>34</sup> Remember, the command `break` will exit the current loop you are in.

#### Daisy population dynamics

In the published Daisy World paper, the population dynamics are written in terms of time:

$$dF_w/dt = F_w \cdot (x \cdot \beta_w - \gamma)$$

$$dF_b/dt = F_b \cdot (x \cdot \beta_b - \gamma)$$

and hence in the form:

$$\frac{dx}{dt} = f(x)$$

Hence we can construct the model via:

$$F_{(t+\Delta t)} \approx F_{(t)} + \Delta t \cdot F_{(t)} \cdot (x \cdot \beta - \gamma)$$

i.e. at each successive time-step, we take the previous fraction ( $F_{(t)}$ ) and add to this, our approximated (forward in time differencing) change in fractional area value.



# 7

## *Numerical modelling – Dynamic (time-stepping)*

ALL MODELS ARE WRONG, BUT SOME ARE USEFUL as the saying goes. Which is actually pretty unfair, as numerical models, in deliberately approximating some aspect of the Real World, are in fact *a priori* designed to be wrong; just sufficiently not wrong, to be useful.

*Forward-in-time (Euler) finite differencing*

Commonly in numerical models, you find that the underlying equations may be of the form:

$$\frac{dx}{dt} = f(x)$$

i.e. the rate of change of some variable  $x$ , is some function of itself ( $x$ ).<sup>1</sup>

Invariably, we wish to make a projection of the state of the system (the value of  $x$  in this example) at some point in the future. If the increment in time is  $\Delta t$ , then we wish to know the value of  $x$  at time  $t + \Delta t$ , i.e.  $x_{(t+\Delta t)}$ .

There is a Taylor expansion for this ... and switching to partial derivative notation, we can write:

$$x_{(t+\Delta t)} = x_{(t)} + \Delta t \cdot \frac{\partial x}{\partial t} + \frac{\Delta t^2}{2} \cdot \frac{\partial^2 x}{\partial t^2} + \frac{\Delta t^3}{6} \cdot \frac{\partial^3 x}{\partial t^3} + O(\Delta t^4)$$

where  $O(\Delta t^4)$  represents 4th order (and smaller) terms (which can be considered as an 'error' term (if not accounted for explicitly)), that will be smaller in magnitude than  $\frac{\Delta t^3}{6} \cdot \frac{\partial^3 x}{\partial t^3}$ .

If we drop all the higher order terms, and solve for  $\frac{dx}{dt}$ , we get:

$$\frac{\partial x}{\partial t} = \frac{x_{(t+\Delta t)} - x_{(t)}}{\Delta t} + O(\Delta t^2)$$

which is just saying that we can approximate (if we accept the error in the approximation represented by  $O(\Delta t^2)$ ) the gradient  $\frac{\partial x}{\partial t}$  (or  $\frac{dx}{dt}$ ) by the difference between the value of  $x$  at time  $t + \Delta t$ , minus the value of  $x$  at time  $t$ , divided by the increment in time,  $\Delta t$ .

In terms of creating a numerical model and coding it up, our next value of  $x$  in time, can hence be approximated:

$$x_{(t+\Delta t)} \approx x_{(t)} + \Delta t \cdot \frac{dx}{dt}$$

*Coding Euler*

How to implement this in code?

Consider the radioactive decay of an amount of radioactive substance. Assume an initial activity  $A$  (don't worry about what the units of this activity are), and the substance decays such that after 1 day, the new activity is equal to half the original activity. We could write (or you might see given to you):

$$\frac{dA}{dt} = -0.5 \cdot A$$

where  $t$  is time in days.

This simply says: the rate of change in  $A$  with time (days), is equal to minus (because it is decaying rather than growing) 0.5 times its value.

<sup>1</sup> The equations need not be a function of time.

We could also write this in the form:

$$\frac{\partial A}{\partial t} = \frac{A(t+\Delta t) - A(t)}{\Delta t} + O(\Delta t^2)$$

and hence in our model, we know that the value of  $A$  at each successive point in time can be written<sup>2</sup>:

$$A_{(t+\Delta t)} \approx A_{(t)} + \Delta t \cdot \frac{dA}{dt}$$

and hence in the specific example:

$$A_{(t+\Delta t)} \approx A_{(t)} - 0.5 \cdot A_{(t)} \cdot \Delta t$$

or

$$A_{(t+\Delta t)} \approx A_{(t)} \cdot (1.0 - 0.5 \cdot \Delta t)$$

If, in code, we represent the time-step  $\Delta t$  by `dt`, we have:

```
A = A*(1-0.5*dt);
```

and in a loop of 100 steps, and initializing the initial activity to one ( $A = 1$ ), we would write:

```
dt = 1.0;
A(1) = 1.0;
time(1) = 0.0;
for n=1:100,
    A(n+1) = A(n)*(1-0.5*dt);
    time(n+1) = n*dt;
end
```

where we create vectors to store all the values of  $A$  together with the corresponding time  $t$  (`A` and `time`, respectively). Or if you prefer:

```
dt = 1.0;
A(1) = 1.0;
time(1) = 0.0;
for n=1:100,
    A(n+1) = A(n)*(1-0.5*dt);
    time(n+1) = time(n) + dt;
end
```

(incrementing the time counter  $t$  with the change in time  $dt$ , each time around the *loop*) or:

```
dt = 1.0;
A(1) = 1.0;
time(1) = 0.0;
n = 1; for t=dt:dt:100*dt,
    n = n+1;
    A(n+1) = A(n)*(1-0.5*dt);
    time(n+1) = t;
end
```

(these codes are equivalent – in the first, you loop with a counter, and then have to derive actual time, and in the second, you loop in

<sup>2</sup> The approximately equal sign reflects the fact that we have omitted the higher order terms and the equation is not exact.

```
228 str = 'do you like bananas?'
```

time, but then have to keep a counter in order to index the output data arrays). Note that we do not start at zero time in the loop, as the initial conditions correspond to time zero.

Try out a couple (or all 3) of these – adding each piece of code to a new *script* file<sup>3</sup>– and explore different values of  $dt$  ( $\Delta t$ ). Add to your code a plot of the results, including appropriate labels etc. (either plot or scatter).

<sup>3</sup> So 2 script files in total ...

Also – in an additional (new) script file – try coding the storing of the results in the form of a single matrix, rather than 2 vectors. For this, rather than create the array (of zeros) of the correct size at the start, you could try something like the following:

```
dt = 1.0;
data(1,1) = 1.0;
data(1,2) = 0.0;
n = 1; for t=dt:dt:100*dt,
    n = n+1;
    data(2,n+1) = data(2,n)*(1-0.5*dt);
    data(1,n+1) = t;
end
```

where the first column of the array *data* is the time, and the second is activity (replacing *A* in the previous example).

You could also shorten this further, to:

```
dt = 1.0;
data(1,1) = 0.0;
data(1,2) = 1.0;
n = 1;
for t=dt:dt:100*dt,
    n = n+1;
    data(n,:) = [t ( data(n-1,2)*(1-0.5*dt) )];
end
```

Try running your decay model (any of the 3 different variants that you have coded up) with different values for the time-step – try much shorter and also much longer than the default value of 1.0 assumed in the codes given to you above. What happens? Why? (View the results graphically to answer this.)



*Other simple finite differencing schemes*

We can also write the Taylor expansion as:

$$x_{(t-\Delta t)} = x_{(t)} - \Delta t \cdot \frac{\partial x}{\partial t} + \frac{\Delta t^2}{2} \cdot \frac{\partial^2 x}{\partial t^2} - \frac{\Delta t^3}{6} \cdot \frac{\partial^3 x}{\partial t^3} + O(\Delta t^4)$$

This leads to the backwards difference operator:

$$\frac{\partial x}{\partial t} = \frac{x_{(t)} - x_{(t-\Delta t)}}{\Delta t} + O(\Delta t^2)$$

Subtracting the second expansion from the first, leads to:

$$\frac{\partial x}{\partial t} = \frac{x_{(t+\Delta t)} - x_{(t-\Delta t)}}{2 \cdot \Delta t} + O(\Delta t^3)$$

which unlike the forwards and backwards operators, is 2nd order accurate. This is known as the centered difference operator. Effectively, it is just saying that the gradient of the function at time  $t$  ( $\frac{dx}{dt}$ ), can be approximated by the average of the gradient between time  $t$  and time  $t - 1$ , and between time  $t$  and time  $t + 1$ .

### 7.1 Catch the ball (ballistics and simulating trajectories)

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 (or simulate numerically)  $d$  – the horizontal distance (in units of  $m$ ) that the ball travels before it hits the ground (a height,  $h$ , of zero). The initial conditions are:

1. The ball is thrown from an initial height  $h_0$  ( $m$ ).
2. The ball is thrown with an initial speed  $s_0$  ( $ms^{-1}$ ).
3. The ball is thrown at an initial angle  $\phi$  with respect to the horizontal.

To keep things simple, we'll neglect any air resistance or spin imparted to 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 and hence that it has zero radius :o)<sup>4</sup>. Over and above this, you'll only need to know the gravitational constant (i.e. gravitational acceleration):  $g = 9.81ms^{-1}$  (i.e. in our example, the ball is being thrown on an Earth-like planet at sealevel).

To simplify things and the construction of the code and encapsulation of the physics of the model, we'll break it down into 4 steps:

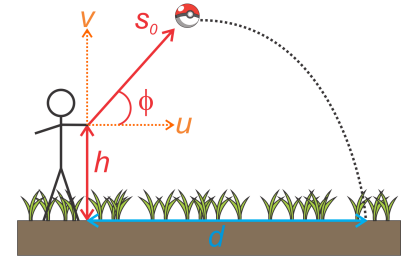


Figure 7.1: Schematic of the thrown-ball system.

<sup>4</sup>How are you going to find a ball with zero radius in the long grass ... ?

*Part I* Considering only horizontal travel.

*Part II* Considering only vertical travel.

*Part III* Considering both horizontal and vertical travel and testing for when the ball hits the ground.

*Part IV* Add some graphical output.

**Part I** Start with a new *script* m-file. For the structure of the code – Figure 7.2 is given as an example to guide you. First, you are going to need to define some constants ( $g$ ), parameters (the initial height  $h_0$ , initial speed ( $s_0$ ), initial angle ( $\theta$ ) of the ball) – these will all go near the start of your program (see Figure 7.2).

Because you are going to use a time-stepping approach (rather than solve the system analytically), you are going to need to create a loop in time, starting at time zero. Can you guess the time-step you will need? No? Then we need to define a variable to contain the value of the time-step (i.e. a *parameter*) that we can then simply

```

ball_u.m
% fun with
ballistics I
< define model
constants
< set model
parameters
< initialize:
* p
* u (calculate)
< calculate
Δp
< update p
< display
variable
values
t
loop
    
```

Figure 7.2: Schematic of the code for simulating the horizontal movement of a ball.

change later. (We might be interested in testing different values of the time-step to ensure that the system is being solved appropriately – i.e. sufficiently accurately and without numerical instability.) You could call this parameter e.g.  $dt$  (for  $dt$ ) and set it<sup>5</sup> to an initial (guessed) value 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 be. 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. Maybe pick a value of 100s to start with.

To recap – as per Figure 7.2 and the instructions above – at this point, as well as suitable comments<sup>6</sup> in your *script* file, you should have:

- 1 variable, assigned with the value for the gravitational constant.
- 3 variables, assigned with values for the initial conditions of the model – the initial height  $h_0$ , initial speed ( $s_0$ ), initial angle ( $\theta$ ) of the ball.
- 2 variables controlling the model numerical solution – the duration of the time-step ( $s$ ), and the total simulation time (also  $s$ ).

Assuming in the example here (you can use something else if you like), 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 = dt:dt:max_t
    %SOME CODE
end
```

with time  $t$  starting at zero, and progressing to  $max\_t$  in steps of  $dt$ . Again – because we already know the ball's position at time zero, we need the loop to start one time-step forward (from zero).

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 after the parameter section of variable definitions, 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.<sup>7</sup> You can calculate the (initial) horizontal component of velocity from the given initial conditions of initial speed ( $s_0$ ) and initial angle of trajectory ( $\phi$ )<sup>8</sup>. For now, pick any 'reasonable' values for  $s_0$ <sup>9</sup> and  $\phi$ <sup>10</sup>. In the figure, the horizontal velocity component is designated  $u$ .

<sup>5</sup> In the parameter definition section of the code.

<sup>6</sup> e.g. you could include the units associated with each variable in the comment line(s) description, so you don't forget ...

<sup>7</sup> 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).

<sup>8</sup> Just as long as you can remember how to calculate the sides of a right angled triangle given the length of the hypotenuse, which here is the speed ... REMEMBER that **MATLAB** uses *radians* for calculating with angles, not *degrees*.

<sup>9</sup> 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.

<sup>10</sup> 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.

Along with the schematic of the code structure, this should be all you need to create a basic code.<sup>11</sup> Check that it runs without error even though it is doing nothing useful yet! 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`.<sup>12</sup>

Now to add some code to the loop. During 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?<sup>13</sup> You need to add this increment in distance travelled to the current value of the position variable  $p$ <sup>14</sup>. Do this – i.e. write down on paper first, if you like, the equation for the change in horizontal position during the time-step, and then write this as code. Update the horizontal position. See margin for hints ...

Re-run the code. Check that 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
    % (YOUR CODE TO UPDATE POSITION GOES HERE)
    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 – here a single long string is formed out of time (number value converted to a string) and position (converted to string), each with a label preceding it.)

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. You can copy your previous m-file and use it as a starting point for the new model<sup>15</sup> or start a fresh *script* m-file – your call.

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/or add to the code. To start with, you should have noticed that the initial vertical position ( $p_{(0)}$ ) of the ball, does not start at zero, but rather at height  $h_{(0)}$  (see Figure 7.1). This is one change to make in the code (i.e. having defined  $h_{(0)}$  as a parameter, you subsequently use  $h_{(0)}$  to set the initial value of  $p$ ). Also – the initial velocity component,  $v$ , is different from before. Go back to your triangle trigonometry, and adjust your so that you are calculating the initial vertical rather than horizontal velocity component. 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.

<sup>11</sup> To recap once more:

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 length. You also 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 before the loop starts, and (horizontal) velocity component  $u$ , which you should have initialized calculated from  $s_0$  and  $\phi$ . There should be nothing in the loop so far.

<sup>12</sup> 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`.

<sup>13</sup> Distance = velocity times time:

$$dp = u \times dt$$

<sup>14</sup> i.e. with code like

```
p = p + delta_p;
```

which you have seen endless times before now and should becoming wearily familiar ...

<sup>15</sup> So for instance we will now interpret the variable  $p$  as the vertical, not horizontal position of the ball.

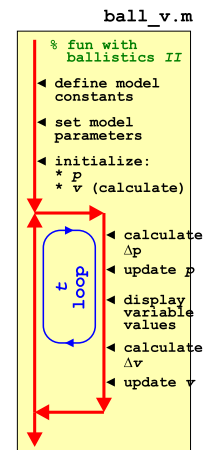


Figure 7.3: Schematic of the code for simulating the vertical movement of a ball.

You could, and indeed should, test the code so far. It should in fact do something very similar to before, with position  $p$  (now interpreted as vertical) 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  $\theta$  you assumed.<sup>16</sup>

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<sup>17</sup>. 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, as we per for position, the change in velocity,  $\Delta v$  as:

$$\Delta v = -\Delta t \cdot g$$

where  $g$  is the acceleration due to gravity and as before,  $\Delta t$  is the time-step duration. Note the appearance of a minus sign in the equation as we are considering a coordinate system with distance upwards and acceleration is downwards.

In the loop add code to calculate the change in velocity during the time-step, and then update the value of  $v$ <sup>18,19</sup>. (Pay attention to the margin notes and program schematic if you are stuck in either the equation, or where to put the code ...)

Re-run the model ... what happens? Does this seem at all '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 (assuming plausible values in each case »») and get a feel for what is the longest duration (in  $s$ ) that the ball lasts in the air (i.e. until its position becomes negative) – this value will guide your choice of a more practical maximum time limit in your loop definition (remembering that you defined the loop limit in time as a parameter near the start of your program).

<sup>16</sup> What value of  $\theta$  would result in an identical change in  $d$  with time (comparing the previous horizontal-only model with the new vertical (only one)?)

<sup>17</sup> Before or after the updating the position? Actually, a slightly tricky question.

<sup>18</sup> 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

<sup>19</sup> Note that in this example and as per Figure 7.3, we update the vertical position in the loop first (at the start of the loop), and then update the velocity afterwards.

**Part III** By this point, you should now have 2 working models (and 2 separate **m-files**) – one for the horizontal position of the ball, and one for the vertical position (and vertical velocity) of the ball. You now need to combine the 2 separate parts of the model into a single (new) program (e.g. named ball\_uv.m).<sup>20</sup>

How to merge? Mostly, the code content of the 2 individual models is identical. What you do need to copy across from the horizontal-only to the vertical-only 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 (separate) models, the variable  $p$  was used to represent both the horizontal AND vertical position of the ball. D'uh!

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, 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 ( $\Delta p$ ) the same if you wish (as this is a temporary variable whose value is replaced each time around the loop in any case), e.g.

```
dp = dt*u;
p(1) = p(1) + dp;
dp = dt*v;
p(2) = p(2) + dp;
```

just as long as you comment your code and do not get the re-use of the same variable name  $dp$  mixed up.

Hopefully this works and runs ... Maybe add some output within the loop to track its progress, such as:

```
for t = 0:dt:max_t
    % (YOU CODE TO UPDATE POSITION)
    disp(['(', num2str(p(1)), ', ', num2str(p(2)), ' ...
        ') @ t = ', num2str(t)]);
    % (YOUR CODE TO UPDATE THE 2 VELOCITY COMPONENTS)
end
```

<sup>20</sup> 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 (i.e. so you end up with 3 different **m-files** by the end of the exercise).

**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 and which is why the model did not work – duh!"

You should end up with output, depending on how you constructed the string to be displayed by `disp` (and what initial conditions you chose ...), looking 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 we want of  $d$  – the distance travelled horizontally when the ball hits the ground.

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 statement (`if ...`) to test whether the vertical position has dropped below zero. This would go in the loop just after the position of the ball (both `p(1)` and `p(2)` components) has been updated, And ...
2. The **MATLAB** command to exit a loop, which you have seen before (look it up if you have forgotten).

Now you might note that when the ball reaches the ground (technically: its height falls below zero) and the loop exits, 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 meters) 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  $\Delta t$  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.  $10\text{cm}$ ?<sup>21</sup> What about  $1\text{cm}$ ?

Finally – as an alternative to creating a `for` loop, in which a maximum number of time-steps (or maximum time) were pre-defined and then having to exit the loop once the ball reaches zero height about the ground, in a new **m-file**<sup>22</sup> – try re-writing the *loop* as a `while` loop, with the condition (for the *loop* to continue looping) that the ball has a height above the ground that is greater than zero. (This makes for a neater solution to the problem.) You might tackle this by first defining a variable (before the loop starts) to store whether the simulation is running or not, e.g.

```
running = true;
```

then the loop will start:

```
while running
```

and you'll test for the ball falling below zero height and set `running` to `false` if so. Because you are not looping through time any longer, you'll need to keep track of time – creating a time variable and setting it to zero (before the loop), and then updating time within the loop, e.g.

```
t = t + dt;
```

---

#### *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, rather than saving all the data (time and position) and then plotting at the very end (after the loop)<sup>23</sup>, are going to plot each new position of the ball as it is calculated, all within the loop.

Take your previous code (either your 3rd script – `ball_uv.m` or the variant where you substituted the `for` loop for a `while` loop) – copy and rename the **m-file**, and then open a new graphics figure window, before the loop starts, and set `hold on`, by adding the lines

```
figure;
hold on;
```

Within the loop, you want to plot each  $(x, y)$  position as it is calculated (i.e. after the horizontal and vertical positions have been updated) by:

<sup>21</sup> i.e. to have the loop terminate when the height is no more than  $-10.0\text{cm}$ .

<sup>22</sup> Copy your 3rd script for calculating both horizontal and vertical position, rename it, and modify the *loop*.

<sup>23</sup> Although if you stored the position of the ball at each time-step, you could re-play the trajectory afterwards.



```
scatter(p(1),p(2));
```

(feel free to add additional parameters to `scatter` to make the points smaller or larger, or filled, or whatever).

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<sup>24</sup>.

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 out the calculation and plot the current position PLUS whatever additional time you tell it to chill out for. The **MATLAB** command (a *function*) is called `pause` and you might initially try in your code:

```
pause(0.05);
```

which should insert a 50ms delay into the loop. Add the pause either immediately before, or immediately after, `scatter` (it doesn't matter which). Run what you have so far.

Now it has all got really trippy. Clearly, if you tell it nothing else, **MATLAB** will insist 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 starts and after you create the figure window, why not prescribe some axes limits(?)<sup>25,26</sup> Having played with the model previously, you should have a good feel for what the maximum vertical and horizontal distances are achieved associated with 'reasonable' choices for the initial conditions ( $s_0$  and  $\theta$ ).

You should end up with something looping like Figure 7.4-ish.

You can have all the trajectories appearing on the same plot of multiple runs of you model, if you comment out the `figure` command in your script, and open a single new figure window at the command line (`>> figure`) before running the model for the first time. Then, each and every time you run the script, the new trajectory will be added on top of all the previous ones. You could also try turning your script into a function so that you do not need to edit the values of  $s_0$  and  $\theta$  in the code to change the trajectory, but instead pass them into the program as parameters instead (the function needs not return anything however), e.g. allowing you to type:

```
>> run_ball_model(10.0,pi/4);
```

(for a model with an initial speed of  $10\text{m}2^{-1}$  and an initial angle from the horizontal of  $\pi/2$  radians).

<sup>24</sup> This is a Trump true fact. In truth, **MATLAB** is about the slowest piece of \*\$&% about.

#### 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.

<sup>25</sup> Remember: the *function* is axis

<sup>26</sup> And also add axis labels ...

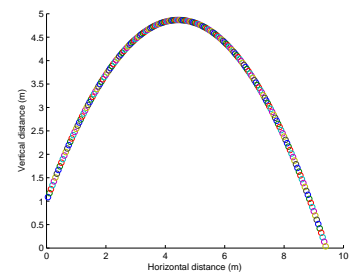


Figure 7.4: Trajectory of a ball!!

Having developed some visualization for the trajectory of the ball, this is a good point to experiment with the length of the time-step and determine at what point (time-step duration) the numerical approximation starts to break down – i.e. as compared to a simulation with a very short time-step (or an analytical solution), when (what longer time-step duration) does the trajectory start to visually differ (and the distance travelled before the ball hits the ground, change)? For example, Figure 7.5 illustrates a 0.1s time-step and Figure 7.6 a 0.2s time-step (contrast with Figure 7.4).

If you are keen ... you can make more of an 'animation' out of the ball trajectory plotting. One trick would be to re-plot the position of the ball a second time, but now in white (hence covering up the previous drawing). But a better strategy is to politely ask **MATLAB** to delete the last plotting (ball) object.

`scatter` is actually a *function*, and a *handle* is returned that is the ID of the points plotted (you don't see or need to worry about this when just typing `scatter` and not assigning the result to anything). You can use this ID to delete the plotting point! e.g. close all the currently open windows and try typing the following \*at the command line):

```
» h=scatter(1,2);
```

and you get a circle plotted at location (1,2). Now try:

```
» delete(h);
```

... and ... it is gone (but leaving (re-scaled) axes and Figure window in place).

If, in your *loop*, after updating the position of the ball, you write (in place or whatever pause and `scatter` code lines you had before):

```
pause(0.025);
delete(h);
h=scatter(p(1),p(2),50,'filled', ...
    'MarkerFaceColor',[1 0 0], 'MarkerEdgeColor',[0 0 0]);
```

you should see a red ball (with a black outline) smoothly sailing across the screen (but see margin notes ...).<sup>27</sup> <sup>28</sup>

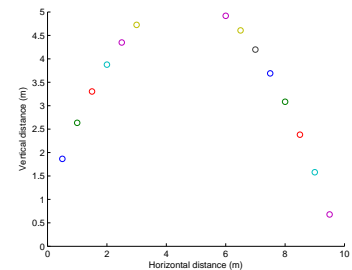


Figure 7.5: Trajectory of a ball (with a poor time-step choice).

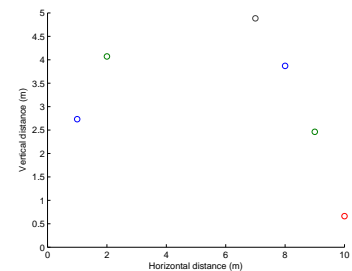


Figure 7.6: Trajectory of a ball (even poorer time-step choice).

<sup>27</sup> You could make the animation a little smoother by decreasing the time-step and also playing about with the delay (pause).

<sup>28</sup> A slight complication here as that as it stands, the code will not work because in the first time around the loop, when you get to `delete(h)`, **MATLAB** is unhappy because the *handle* `h` has not yet been defined anywhere. The easiest way to fix this is outside the loop, to plot the initial position of the ball (and obtain its *handle*). e.g.:

```
figure;
axis([0 10 0 5]);
hold on;
h=scatter(p(1),p(2),50,'filled');
```

Now when the loop starts, there is a 'ball' to delete!

[OPTIONAL] A further refinement would be to add a term to account for air resistance – as the ball travels through the air, friction will act to decelerate the ball.

You could represent the effect of friction in a similar way to how you accounted for gravity, except (a) friction will affect both velocity components, and (ii) friction will act to decelerate the ball, regardless of its direction of travel (up or downwards). Friction also differs from gravitational acceleration in that the deceleration will not be constant, but instead a function of velocity. Furthermore, can assume that friction will scale with the square of the velocity (rather than linearly).

In your basic code:

```
dp = dt*u;
p(1) = p(1) + dp;
dp = dt*v;
p(2) = p(2) + dp;
dv = -dt*g;
v = v + dv;
```

you would add (to the end of the loop):

```
du = -dt*f*u^2;
u = u + du;
dv = -dt*f*v^2;
v = v + dv;
```

Here,  $f$  is a parameter that scales the impact of air resistance on velocity. It is not clear, at least in this simplistic formulation, what its value should be. So this (the value of  $f$ ) is something to explore and test the effect of.

## 7.2 Dynamics in the zero-D Energy-balance climate model

In this next Example making use of time-stepping, we will 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 physical system as defined in the equations has any heat capacity and the outgoing (long-wave / infrared) energy flux is always assumed to be in exact equilibrium with the incoming (short-wave) 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 that the 2 equations are equal (and hence solve for  $T$ ). 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)<sup>29</sup> to link this 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 will assume the following:

- The average mixed layer depth of the ocean is 70 m.
- The average fraction of the Earth's surface that is ocean is 0.7.

(both from *Henderson-Sellers* [2014]) – Figure 7.7. You'll also need to know:

- The specific heat capacity of water.

(see Box) but you can find this out for yourself ... 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 (i.e. we are considering a representative  $1m^2$  of surface, of which 70% is water (or  $0.7m^2$ ) with a depth of  $70m$ ).

The form of the program is shown schematically in Figure 7.8. 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,

### 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  $\Delta Q$  if you prefer) and  $\Delta T$  the associated change in temperature. Units are:

- $C - JK^{-1}$
- $\Delta T - K$
- $Q - J$

Typical units for specific heat capacity are:

$$Jg^{-1}K^{-1}$$

(or  $Jkg^{-1}K^{-1}$ )

<sup>29</sup> Once again – be very careful with the units. Or all will be lost ...

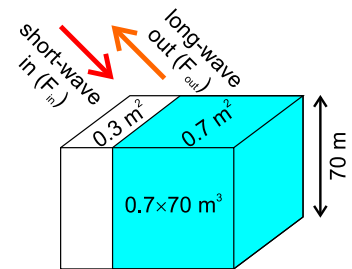


Figure 7.7: Schematic of the dynamic EBM.

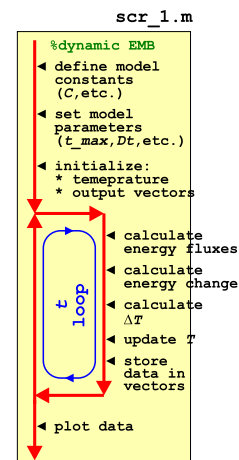


Figure 7.8: Schematic of the script for the basic dynamic EBM

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 loop time increment,  $\Delta T = 1.0$  (year). 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 one) between incoming solar radiation absorbed and out-going infrared radiation loss. For this – taken the equations given to you earlier for absorbed solar radiation and infrared loss, and simply calculate the difference (rather than re-write in terms of  $T$  as you did for the equilibrium EBM) –  $\Delta F$ .

From the energy flux imbalance ( $\Delta F$ ), which is in units of  $Wm^{-2}$ , i.e.  $J s^{-1} m^{-2}$ , you'll need to calculate how many  $J$  of energy (per  $m^2$ ) are lost or gained over the course of your time-step. Your time-step is in units of years ... so you'll need to calculate how many  $s$  in a (average) year, and multiply the energy change  $s^{-1}$  by this number (to give the energy change per time-step). The energy change can then be used to update the temperature of the mixed layer ocean ... as long as you have already calculated the heat capacity of the ocean that is ...<sup>30</sup>.

A possible sequence of calculations (assuming you have calculated the heat capacity of the ocean box once, before the loop starts) follows<sup>31</sup>:

1. Calculate incoming energy flux,  $F_{in}$ .
2. Calculate outgoing energy flux,  $F_{out}$ .
3. Calculate the net energy change (per  $m^2$  per  $s$ ) at the Earth's surface,  $\Delta F$ .
4. Calculate the total energy imbalance (per  $m^2$ ) over a year, in  $J$ .
5. Using the heat capacity of the 'ocean', calculate its temperature change.

After the loop, plot something helpful at the end. If successful, you should see something similar to (actually, identical to) Figure 7.9 (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  $\Delta t$ . If  $dt$  is your parameter representing the increment in time (presumably, conveniently defined near the start of the code)<sup>32</sup>, and  $max\_t$  the maximum time (here: 100 years) (also conveniently defined near the start of the code?), then:

```
% start of time-stepping loop
```

<sup>30</sup> Assuming specific heat capacity is in units of  $J g^{-1} K^{-1}$ , you need to find the mass of the ocean box in  $g$ , noting that the density of (pure water at 0C) is  $1 g cm^{-3}$ .

Start by determining the volume of the ocean box in  $cm^3$ , convert to  $g$ , and then multiply the specific heat capacity  $C$  by this, to give the heat capacity of the ocean box.

This is the number of  $J$  of energy needed to raise the temperature by 1K.

<sup>31</sup> It is much easier and less prone to bug, if you split things into five steps.

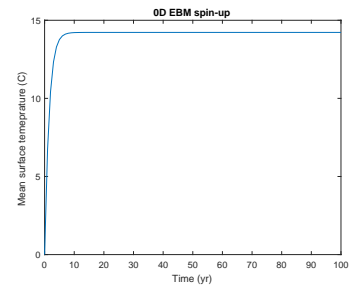


Figure 7.9: 100 yr spin-up of the basic EBM.

```
scr_2.m
%dynamic EMB
< define model
  constants
  (C, etc.)
< set model
  parameters
  (t_max, Dt, etc.)
< initialize:
  * temperature
  * output vectors
  * loop counter
  < loop
    < increment
      loop counter
    < calculate
      energy fluxes
    < calculate
      energy change
    < calculate
      ΔT
    < update T
    < store
      data in
      vectors
  < plot data
```

Figure 7.10: Schematic of the script for the basic dynamic EBM – now with added loop count(!)

<sup>32</sup> Don't forget to convert  $dt$  into units of  $s$  when you use it in the energy calculation.

242 str = 'do you like bananas?'

```
for t = dt:dt:max_t,  
    % SOME CODE GOES HERE  
end
```

Now you will need to create yourself a loop counter in order to store the results (for subsequent plotting), because `dt` will not necessarily be an integer and hence you will not be able to use `t` to index your data storage vector (/array). The modification needed is only minor however – see Figure 7.10.

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 7.8 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(100,1);  
data_T = zeros(100,1);
```

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 7.8 rather than Figure 7.10).

By playing around with different parameter values for  $\Delta t$ , you should discover that some care has to be taken with the choice of time-step duration, e.g. Figure 7.11 has a time-step of 3.5 years, which clearly is on the verge of going doolally.<sup>33</sup>

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 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  $\text{CO}_2$ , 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 and calculates the warming impact of a prescribed  $\text{CO}_2$  concentration (technically: mixing ratio) in the atmosphere.

First off: copy either of your previous dynamic EBM scripts (`scr_1`, `scr_2`), re-naming to e.g. `scr_3`.

Then, check out the  $\text{CO}_2$  radiative forcing (Greenhouse Effect) Box. This will guide you as to how you are going to modify your energy

<sup>33</sup> 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.

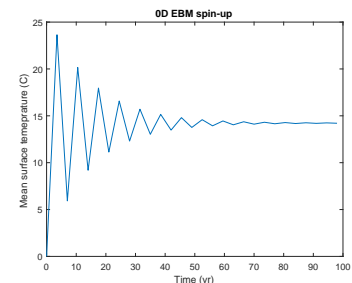


Figure 7.11: 100 yr spin-up of the basic EBM, but with a poor choice of time-step ...

### The Greenhouse Effect

The effect of changing  $\text{CO}_2$  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,  $\Delta F$ , as a function of  $\text{CO}_2$  concentration (technically: mixing ratio) ( $\text{CO}_2$ ) relative to a reference (pre-industrial) concentration (typically:  $\text{CO}_{2(0)} = 278 \text{ ppm}$ ) can be approximated:

$$\Delta F = 5.35 \cdot \ln\left(\frac{\text{CO}_2}{\text{CO}_{2(0)}}\right)$$

The complete basic EBM energy budget now looks like:

$$F_{in} = \frac{(1-\alpha) \cdot S_0}{4} + 5.35 \cdot \ln\left(\frac{\text{CO}_2}{\text{CO}_{2(0)}}\right)$$

$$F_{out} = 0.62 \cdot \sigma \cdot T^4$$

budget (within the time-stepping loop) – basically, you are simply adding a 3rd term (a second incoming term) to the heat budget.

From your previous experiments, you should have determined what value the equilibrium temperature ended up as (in the absence of CO<sub>2</sub> forcing and with a modern solar constant). You should make this your new initial condition for calculating the planetary temperature from 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 cleanly responding to whatever perturbation you have applied (/experiment carried out).)

Test the model with a fixed, assumed CO<sub>2</sub> concentration (by setting the value of your parameter for CO<sub>2</sub> concentration) and check that the mean surface temperature responds in a reasonable way.<sup>34,35</sup> For reference:

- Peak of last glacial —  $\sim 190\text{ppm}$
- Pre-industrial —  $278\text{ppm}$
- Current —  $\sim 400\text{ppm}$
- End of century —  $\sim 900\text{ppm}$
- Cretaceous —  $\sim 834 - 1112\text{ppm}(?)$

NEXT, you will load in a CO<sub>2</sub> data-set and drive your dynamic zero-D EBM as a function of time, with a changing concentration of CO<sub>2</sub> in the atmosphere.

The program (scr\_3) structure is going to be similar to Figure 7.12. To complete it, you need to:

1. Add in code to load in the CO<sub>2</sub> 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<sub>2</sub> value, temperature. (Create a single, 3-column 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. Also in the loop – save the current year, CO<sub>2</sub> value, and associated calculated temperature.

Be careful that indexing of arrays in **MATLAB** (for accessing the CO<sub>2</sub> value, or saving data to the appropriate row in the vector or array) –

<sup>34</sup> What is 'reasonable'? Well, you could conduct a pair of experiments – one in which you do not modify CO<sub>2</sub>, and one in which you double it. The IPCC and there (now) five Assessment reports have much to say about the climate system response to a doubling of CO<sub>2</sub>. So you can conduct a reality check on your model based on existing and widely available climate sensitivity information.

<sup>35</sup> By way of reference: assume that the pre-industrial concentration (mixing ratio) of CO<sub>2</sub> in the atmosphere (CO<sub>2(0)</sub>) is 278 ppm.

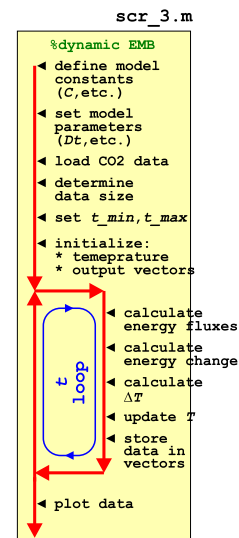


Figure 7.12: Schematic of the dynamic EBM driven by a history of CO<sub>2</sub> (read in from a file).



**MATLAB** always starts at a value of 1. You will either need to derive an index from the current year<sup>36</sup>, 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 7.13 (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. Create 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 7.14.

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.

---

To quantify the impact of the ocean heat reservoir on the transient climate response – try modifying one of your original equilibrium EBM function such that rather than a value of  $S_0$ , you instead pass in the  $CO_2$  concentration. You'll need to add in the  $CO_2$  radiative term to the energy balance equation (see earlier Box on  $CO_2$  radiative forcing) as you solve for  $T$ . Take (and rename) the dynamic EBM script (`scr_3`), and in place of the lines of code in the loop that calculated the radiative imbalance and then updated the global surface temperature – simply call your modified EBM function.

The aim here is to be able to run the same experiment of changing  $CO_2$ , but with the assumption that the climate is always in equilibrium. Compare the equilibrium vs. dynamic model results (giving an estimation of the importance of the non zero heat capacity of the planet in creating a lag in temperature in response to a forcing).

---

A further refinement would be to add a deep ocean heat reservoir (with e.g. diffusive exchange between deep and surface (mixed layer) boxes).

<sup>36</sup> e.g. current year minus start year plus one.

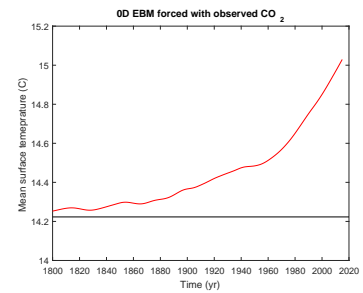


Figure 7.13: Transient EBM response to observed changes in atmospheric  $CO_2$ . For reference, the pre-industrial equilibrium global temperature is shown as a horizontal black line.

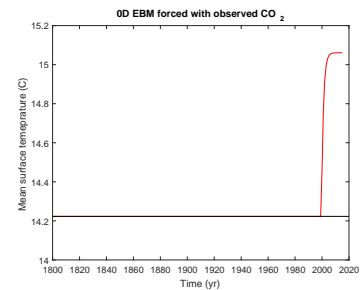


Figure 7.14: Transient EBM response to (fake) changes in atmospheric  $CO_2$ .



8

*Numerical modelling – To infinity (1D) and beyond(!)*

### 8.1 1-D energy-balance climate model

Although the Earth is, of course, fundamentally three-dimensional, there are many situations in Earth, Ocean, and Atmospheric sciences when an environmental system can be approximated with a model having just one single (length) dimension. For instance, the structure (e.g. temperature properties) of the atmosphere generally varies vertically much more quickly in distance than it does in the horizontal. Similarly, the changes in the physical, biological, and chemical properties of the ocean are generally much more pronounced with a change in depth rather than for the same distance in latitude or longitude. Because the horizontal gradients in environmental properties in such systems are often relatively small, the horizontal fluxes and exchanges of matter and energy will also be small, particularly compared to vertical transport. The behaviour of some processes which are in reality are operating in a three-dimensional system world can therefore often be usefully analysed by considering their behaviour in just one dimension.

---

THE SIMPLEST POSSIBLE<sup>1</sup> EXAMPLE of a 1-D model is to build on the (zero-D) EBM from before. Well ... perhaps not the simplest, but relatively fun. If you like that sort of thing ...

The idea is: rather than to treat the entire Earth as a single homogeneous surface characterized by a single surface temperature (and hence single value of outgoing radiation flux), you are going to split the Earth's surface up into latitudinal bands. Why latitude and not longitude? Simple inspection of global temperature distributions indicate that the meridional<sup>2</sup> gradients are much more pronounced than the zonal<sup>3</sup> gradients. Obviously, a model would be improved by resolving both meridional and zonal gradients and energy flows, but if you are going to simplify a climate model to just a single dimension, picking latitude seems as good a way to go any any. You can also think in terms of how incoming solar radiation changes most – ignoring day-night changes as the Earth rotates – low vs. high latitude regions have the greatest contrast in incoming energy (and hence temperature), and one might suspect that flow of (heat) energy from the Equator towards the poles might be about the single most important transport in the climate system.

We can make a further approximation by noting that the input of solar radiation is roughly symmetrical about the Equator (and assuming that we are going to consider only an annual average climate state of the Earth).<sup>4,5</sup> So, for this exercise, you need actually only model one hemisphere (and assume that the other one acts

<sup>1</sup> :o)

#### EXAMPLE OVERVIEW:

1. Define model grid (latitudes)
2. Calculate zonal surface area
3. Calculate zonal cross-sectional area
4. Calculate incident solar radiation
5. Set up plotting as a function of latitude

<sup>2</sup> According to the mighty Wikipedia: "along a meridian" or "in the north-south direction".

<sup>3</sup> "along a latitude circle" or "in the west-east direction"

<sup>4</sup> The actual distribution of the continents on Earth together with how the ocean then circulates on a large-scale completely ruins in this assumption practice, or rather: should a particular degree of 'realism' be required.

<sup>5</sup> Because of the (non-zero) obliquity of the Earth, there is a slightly imbalance in the annual averaged solar radiation received by each hemisphere – dictated by which hemisphere is in its summer when the Earth is closest to the Sun.

identically and that the resulting temperature distribution can be copied/mirrored).

OK – so the first step is to divide up the Earth (or one hemisphere), into bands, with each band being subject to the same energy budget as before, including an ocean-dominated heat capacity component, and which will lead to each band having its own characteristic temperature. (Assume for now that each latitude band is characterized by the same fraction of ocean and mean mixed-layer depth.) You can choose how many bands to make. Actually, if you do it the ‘easy’ way it will not matter how many you want<sup>6</sup> and which, as you might have guessed, uses loops. The hard way is to write out all the equations explicitly<sup>7</sup>.

You are going to do construct something like this:

```
for n = 1:n_max
    % CODE GOES HERE
end
```

where  $n\_max = 90.0/dlat$  and  $dlat$  is the width of each band<sup>8</sup>.

For each band, it would be nice to write exactly the same equations as before. Except ... you can't. Why? (Hint: spheres have curved surfaces – who would have guessed? And the surface gets more oblique with respect to incoming radiation as the latitude increases, meaning that the same (per unit area) solar flux is spread over an increasing area.)

- For outgoing radiation / energy loss, you need to know the surface area of each band, assuming that each band occupies an equal number of degrees of latitude, and how this varies with latitude. A small hint can be found in Box #1. Or the Internet will, as usual, know it all.
- For incoming solar radiation, you need the cross-sectional area of a band on a sphere.

The original mean incident solar energy per unit area was  $S_0/4$  on the basis that the total received radiation was  $\pi \cdot r_0^2 \cdot S_0$  spread over (i.e. divided by) a total surface area of  $4 \cdot \pi \cdot r_0^2$ . You already have the total surface area of a zonal band around the Earth (Box #1) which you need for calculating the long-wave energy loss from, but now you need the area perpendicular to the incoming solar radiation (i.e. the cross-sectional area). The area of a complete disk is  $\pi \cdot r_0^2$  and to cut a long story short ... and see Box #2 ... the area of a portion of a disk, is:

$$A = \frac{r_0^2}{2} \cdot (-2 \cdot \phi_1 + 2 \cdot \phi_2 - \sin(2 \cdot \phi_1) + \sin(2 \cdot \phi_2))$$

which is \*so\* much less fun than before :(

Actually, both equations are so little fun, that, assuming that you

#### #1 Zonal area of the Earth's surface

The area of a zonal band of the Earth surface, from latitude  $\phi_1$  to  $\phi_2$  (in radians), can be found by integrating the circumference of a circle:  $2 \cdot \pi \cdot r$ , where  $r = r_0 \cdot \cos(\phi)$  and  $r_0$  is the radius of the Earth:

$$\int_{\phi_1}^{\phi_2} 2 \cdot \pi \cdot r_0 \cdot \cos(\phi) \cdot dx$$

and where  $\delta x$  is an increment in length tangential to the surface equal to  $r_0 \cdot \sin(\delta\phi)$  and which for small  $\delta\phi$  as can be written as  $r_0 \cdot \delta\phi$ .

In the limit  $\delta\phi \rightarrow 0$ :

$$\int_{\phi_1}^{\phi_2} 2 \cdot \pi \cdot r_0^2 \cdot \cos(\phi) \cdot d\phi$$

The zonal area between latitude  $\phi_1$  and  $\phi_2$  is thus:

$$2 \cdot \pi \cdot r_0^2 \cdot (\sin(\phi_2) - \sin(\phi_1))$$

and which is why when you integrate from  $-90^\circ$  to  $+90^\circ$  (or  $-\pi/2$  to  $+\pi/2$ ) you recover the surface area of a sphere:  $4 \cdot \pi \cdot r_0^2$ .

<sup>6</sup> Within reason, but ... as you'll find later, there is a numerical stability penalty to having too many (but simply requiring a shorter time-step to fix.)

<sup>7</sup> If you are unsure how a loop is going to pan out in terms of updating the fluxes and calculating the temperature of each zonal band, maybe write out the equations in full initially (for one hemisphere), e.g. for 3 bands: 0-30°N, 30-60°N, and 60-90°N.

<sup>8</sup> If you loop in  $n$  (latitudinal bands), you can pre-define the northern and southern edge of each band for convenience, and then simply by indexing the appropriate array with  $n$ , recover the latitude, e.g.

```
% define model grid - N
edge
grid_n = [dlat:dlat:90];
% define model grid - S
edge
grid_s =
[0:dlat:90-dlat];
```

where  $dlat$  is the increment in latitude between bands.

defined vectors to hold the northern and southern edges of the zonal bands (see later), I'll give you the necessary code fragment for free:

```
% calculate zonal surface area (units radius)
loc_sa = 2.0*pi*( ...
    ( sin(pi*grid_n(n)/180)-sin(pi*grid_s(n)/180) ...
    );
% calculate cross-sectional area
loc_ca = 0.5*( ...
    - 2.0*pi*grid_s(n)/180 + 2.0*pi*grid_n(n)/180 - ...
    sin(2.0*pi*grid_s(n)/180) + sin(2.0*pi*grid_n(n)/180)
    ...
    );
```

where `loc_sa` is the surface area of the zonal band, and `loc_ca` is the cross-sectional area (`grid_n` and `grid_s` hold the northern and southern edges, respectively, of the zonal bands).

Obviously(!) you ratio `loc_ca` by `loc_sa` to get out the relative change in solar flux for that latitudinal zone (as you did for a disk/sphere and ended up with  $S_0/4$ ). Note that **MATLAB** just hates units of  $^\circ$  for angles – you need your latitude values, when you calculate the *sin* of the southern and northern boundaries of the zonal band, in units of radians.

You are going to be time-stepping through the simulation (as per the previous EBM with a heat reservoir), and your time-stepping loop needs to go outside (around) the latitude band ( $n$ ) loop. The 'code goes here'<sup>9</sup> is going to be similar to the code as before, for updating the temperature of the surface (equivalent to the temperature of your ocean mixed layer heat reservoir), but obviously you need a vector to store the temperature of each zonal band.

You are ready to go ... or should be. Probably easiest is to adapt your function from before (and save under a different m-file name) and retain the ability to pass in a time-step and also maximum simulation duration. Amazingly, given the cr\*ppy unpleasant trigonometry involved, it seems to work(!) – illustrated in Figure 8.1. As ever, if you give it a particularly inappropriate time-step, funky and meaningless things can happen (not shown).

---

IN AN EXTENSION TO THIS EXAMPLE, we note that although the distribution of surface temperatures with latitude looks not entirely unreasonable (colder at the poles is good!), you really need data<sup>10</sup> of some sort to be sure the model projection is not bonkers. You had a dataset of annual mean global surface air temperature data before (which you dutifully plotted). You could either eye-ball some numbers from and try and guess appropriate or representative values as a function of latitude and compare to your EBM, or calculate a zonal

## #2 Zonal cross-sectional area

The cross-sectional area of a zonal band ... is a pig to calculate. You start with the area of a circle bordered by a cord, which can be thought of as a line of latitude. This itself, is derived by calculating the area of a segment and subtracting a triangle ... no seriously. I wish I could be bothered to draw you a picture. Google is full of hits for a circular segment.

Inconveniently, this is written in terms of the angle of the segment,  $\psi$ :

$$A = \frac{r_0^2}{2} \cdot (\psi - \sin(\psi))$$

Again, you need a picture. If we re-write  $\psi$  in terms of latitude  $\phi$ :

$$\phi = \frac{(\pi - \psi)}{2}$$

then we can reduce this to (recognising, e.g. that  $\sin(\pi - 2 \cdot \phi)$  is simply  $\sin(2 \cdot \phi)$ ):

$$A = \frac{r_0^2}{2} \cdot (\pi - 2 \cdot \phi - \sin(2 \cdot \phi))$$

All we need to do then, is to subtract the smaller, high-latitude chord-bounded circular segment from the low-latitude one. Simples.

<sup>9</sup> Along the lines of:

```
% (1) calculate net
radiation imbalance (W
m-2)
% (2) update temperature
(of ocean mixed layer)
```

(with the results array having a zonal band number dimension as well as of time).

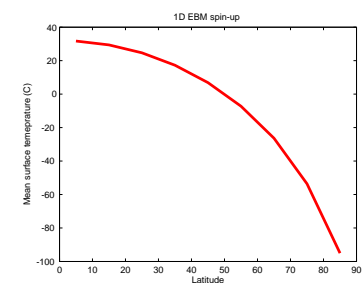


Figure 8.1: Basic 1-D EBM with no latitudinal heat transport and for a single hemisphere only.

<sup>10</sup> Not the Star Trek, Next Generation, one.

mean. Actually, **MATLAB** makes this obscenely simple for you using the mean function<sup>11</sup>.

The only things then to watch out for are:

1. If the array is in the wrong orientation, you'll find yourself averaging along lines of latitude. This is simple to check as you'll get no noticeable latitudinal gradient in temperature. You should also find in that case that the length of the vector returned by mean matches the longitude grid rather than latitude.
2. Correcting #1 requires flipping the matrix around with the transpose operator (').
3. Units – units of the temperature dataset are K whereas your model is in degrees Centigrade.

Once you have fixed any obvious data problems, you should end up with something like Figure 8.2 (January) or Figure 8.3 (July). Still to be done is to create an annual average zonal mean from the data that can be contrasted directly with the annual average EBM output, rather than just a single month of data. Fixing this is left as an exercise for the reader, as they say ...

Irrespective of the month (and this might well hold true for the annual mean too), the EBM doesn't exactly provide an ideal fit to the observations. In particular: the North pole is rather too cold and the tropics maybe a little on the warm side. Actually, we are only really looking at half the model-data picture at the moment, and although in the EBM the Southern Hemisphere is a mirror image of the North, it would help to actually see this. So in addition to creating an annual mean zonal temperature profile to plot against the EBM – also (calculate, or mirror, and) plot the corresponding model projection for the Southern Hemisphere. Something is still missing (in terms of the model accounting for the observations) – what? Hopefully you correctly guessed (i.e. scientifically and logically deduced) that it is meridional heat transport – from the (overly) warm tropics to the (too) cold poles.<sup>12</sup>

---

EXTENDING THIS EXAMPLE FURTHER, we'll add some meridional transport of heat energy (to fix the process missing from the previous version).

We can encapsulate something of the effect of heat transport along the latitudinal temperature gradient, either by adding a term to represent eddy diffusion and analogous to Fick's law, or by analogy to thermal conductance (albeit with a very poorly conducting atmosphere). They actually both amount to the same thing and will end up with similar looking equations. Taking the thermal conductance

<sup>11</sup> A function to calculate the arithmetic mean, rather than a nasty and vindictive function.

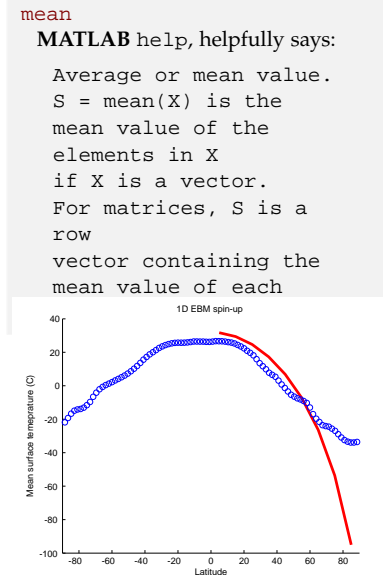


Figure 8.2: Basic 1-D EBM with no latitudinal heat transport (red filled circles). Overlain is the zonal mean observational data for January (blue circles).

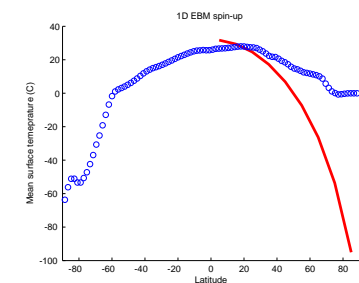


Figure 8.3: As per Figure 8.2 but for July.

<sup>12</sup> We have also ignored e.g. how surface albedo increases as incident angle decreases – i.e. solar radiation is generally absorbed more strongly by surface that are perpendicular to the radiation and reflected more efficiently if radiation is glancing at a shallow angle to the surface. However, this would only exacerbate our problem and leave the poles even colder.

approach, the flux of heat energy from one latitudinal band to the next,  $J$  (W), can be written<sup>13</sup>:

$$J = -k \cdot A \cdot \frac{\Delta T}{\Delta z}$$

where  $k$  is the thermal conductivity ( $Wm^{-1}K^{-1}$ ),  $\Delta T$  is the difference between the temperatures of two adjacent zonal bands ( $T_1 - T_2$ ), and  $\Delta z$  the distance between the bands (measured at the mid-point latitude<sup>14</sup>).<sup>15</sup>

To code this, you simply take the interface area between two adjacent zonal bands ( $A$ ), multiplied by  $k$ , and by the temperature gradient between the bands ( $\frac{\Delta T}{\Delta z}$ ). Heat energy will be lost by the band with the higher temperature, and gained by the adjacent band with the lower temperature, which needs to be taken into account in the energy budget of each band, as summarized below.

The area that heat diffuses across can be simply approximated as the height of the atmosphere over which heat transport takes place, multiplied by the distance around the Earth at that latitude (taking the latitude at the boundary between zonal bands, rather than the mid-point). We'll further assume that for height, the atmosphere can be approximated by equivalent thickness of constant pressure, which would make it 8.5 km (8.5E6 m) in height (and then suddenly space beyond that).

Based on the equation – add a heat diffusion (/conductance) term to your 1D zonal EBM. Note that you do not *a priori* know the value of  $k$ . This is not a problem *per se*, indeed, there may be no simple answer or first principals derivation because the processes that govern meridional heat transport in the real atmosphere ... and ocean, may be legion and non-linear. The advantage of a model is that you can find a value of  $k$  that most closely fits the observed data and thus best represents the missing process. Informally, you can simply play with the model and by trial-and-error find a value that seems to fit the observations best.

The key here is to recognise that there are now additional terms in calculating the energy balance for any particular zone. Whereas previously we could write:

$$\Delta F_{(n)} = F_{solar\_in(n)} - F_{longwave\_out(n)}$$

now we need:

$$\Delta F_{(n)} = F_{solar\_in(n)} - F_{longwave\_out(n)} + F_{diffusion\_in(n)} - F_{diffusion\_out(n)}$$

Note that we have special boundary conditions to consider: the zone bordering the Equator and the zone bordering the pole. This is because the polar zone only gains heat by diffusion from lower latitudes and there is no higher latitude zone than it to diffuse heat to.

<sup>13</sup> The equation is conventionally written as negative, assuming the point of reference is the higher temperature, which loses heat energy.

<sup>14</sup> Similar to before, if you loop in  $n$  (latitudinal bands), you can pre-define the central latitude of each band for convenience:

```
% define model grid
mid-point
grid_mid = ...
[0+dlat/2:dlat:90-dlat/2];
```

although ... this comes in useful only for plotting (e.g. temperatures against the mid-point latitude of the zonal bands, as the separation in latitude is always  $dlat$  and hence the separation in distance is always the same(!)).

<sup>15</sup> This is effectively the same as for the diffusion of  $CH_4$  in a soil column in the other 1D modelling example, with the exception of the addition of an explicit area ( $A$ ) term here, which we did not worry about before because the model was constructed on a unit area ( $1\text{ cm}^2$ ) basis and hence area did not appear explicitly in the equations.

#### Distance between 2 latitudes

Really, you don't need a Box for this. It is embarrassing to make one in fact. But just in case ...

The average distance between zonal bands can be estimated from the difference in latitude between the two mid-points of the zones, and divide up the circumference of the Earth proportionally, i.e.

$$\Delta z = \frac{\Delta lat}{360} \cdot z_{total}$$

where  $z_{total} = 2 \cdot \pi \cdot R$  (the circumference of the Earth at the Equator).

#### Circumference at a specific latitude

This is even more embarrassing to write than the last one. The distance,  $z$ , around a particular latitude,  $\phi$  (a Greek character was really not necessary, but it looks way more fancy this way), is:

$$z = 2 \cdot \pi \cdot \sin(\phi) \cdot R$$

( $\sin(\phi) \cdot R$  being the radius of the circle at that latitude).

For the lowest latitude zone, if we are assuming that the Earth is symmetrical about the Equator, then it only loses heat to a higher latitude zone and does not exchange heat energy with the opposite hemisphere (because the temperature is assumed the same).

The structure of your model, within the (outer) time-stepping loop, should then look like:

1. Loop through all  $n$  latitude bands and calculate the in-coming and out-going radiation.<sup>16</sup>
2. Loop through  $(n - 1)$  latitude bands (i.e. omitting the highest latitude box,  $n$ ), and calculate the diffusion of heat from the band  $n$  to the one adjacent at higher latitude  $(n + 1)$ . Populate 2 (length  $n$ ) vectors – one to store the diffusive heat gain (presumably from a lower latitude), which will have non-zero values for indices 2 through  $n$ , and one to store the diffusive heat loss (presumably to a higher latitude), which will have non-zero values for indices 1 through  $(n - 1)$ .
3. Loop through all  $n$  latitude bands, calculate the net energy input  $\Delta F_{(n)}$  and update the surface temperature accordingly (based on the heat capacity of the ocean mixed layer and the time-step, as before).

What about the value of  $k$ ? You are going to have to guess it to begin with<sup>17</sup> ... and adjust your guess if the model fits the data worse than before.

As an illustration – Figure 8.4 shows the effect of specifying a value of heat conductivity of  $k = 0.1 \text{ Wm}^{-1}\text{K}^{-1}$ , while  $k = 1.0 \text{ Wm}^{-1}\text{K}^{-1}$ , as shown in Figure 8.5, is clearly complete overkill, and much of the pole-to-Equator temperature gradient has been wiped out by over-aggressive heat transport between the bands. (Note that here I have simply mirrored the modelling temperature profile for the Northern hemisphere, to the other (with a hold on). This could have been done much better by combining the vectors and hence obtaining a continuous curve from Southern to North.)

<sup>16</sup> Don't update any temperatures just yet!

As before, if you are not entirely confident in what you are doing – write out the equations long-hand for the simplest possible comparable case – that of 3 zonal bands: one from 0-30°N, one 30-60°N, and one from 60-90°N. You have two flux calculations in this case – the transfer of heat energy from the low to the mid latitude box, and from the mid to the high latitude zone. See if you can see the pattern, which will then help you generalize it to  $n$ .

<sup>17</sup> If you see nothing plotted, your guess might be too large and you have numerical instability. You could try reducing the time-step. But also start with the lowest conceivable value and work higher.

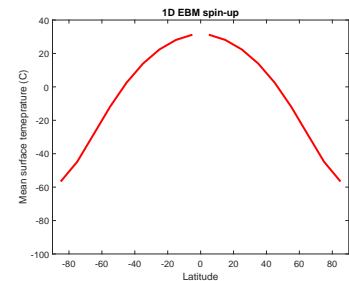


Figure 8.4: 1D EBM with an initial guess as to the value of  $k$ .

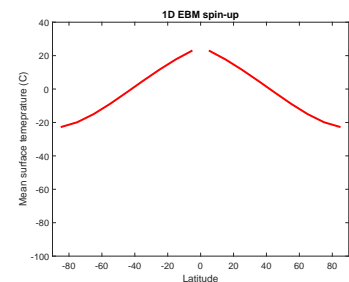


Figure 8.5: 1D EBM with a  $\times 10$  larger value of  $k$ .

## 8.2 1-D reaction-transport model

A RATHER SCIENTIFICALLY DIFFERENT, BUT CONCEPTUALLY SOMEWHAT SIMILAR EXAMPLE, consider diffusion of a gas through a porous medium. We will take the example of methane ( $\text{CH}_4$ ) diffusion into soils, but there are many other situations in the Earth, Ocean, and Atmospheric sciences where (diffusive) transport in 1-D is critical to understand (such as the supply of solutes to the interface of a growing mineral crystal). At its simplest, we have a concentration of  $\text{CH}_4$  in the atmosphere, which we will assume does not change with time (i.e., the reservoir is in effect infinite). We will call this concentration  $C_0$ . Because we are not going to allow the value of  $C_0$  be affected by whatever happens in our 1-D soil column (we are not concerned in this exercise in any role that the soil methane sink might play in controlling the concentration of  $\text{CH}_4$  in the atmosphere itself), it is a condition imposed on the model. This is known as a boundary condition (and because it is at the top of the soil column, it is an upper boundary condition).

In the soil we have a population of methane-consuming bacteria ('methanotrophs') who are taking up and metabolizing the  $\text{CH}_4$  (there will also thus also be a return of  $\text{CO}_2$ , the metabolic product of  $\text{CH}_4$  oxidation, from the soil to the atmosphere). Because  $\text{CH}_4$  is being depleted at depth, there will be a gradient in  $\text{CH}_4$  concentrations along which  $\text{CH}_4$  there will be net diffusive transport, illustrated in Figure 8.6. The scientific question is thus; what is the flux of  $\text{CH}_4$  into soils? This is important (no, really!) because methane is a powerful greenhouse gas and (aerobic) soils might constitute an important sink of this gas.<sup>18</sup>

If all  $\text{CH}_4$  in the pore space was entirely consumed at some known depth,  $z$ , then we would have a gradient of  $C_0 - 0$  ( $C_0$  being the imposed upper boundary condition, and zero being the concentration at depth) in methane concentration, and diffusion would be taking place over a depth  $z$ . If  $D$  is the diffusivity of  $\text{CH}_4$  (in soil), with units of  $\text{cm}^2\text{s}^{-1}$ , then we can easily calculate the initial flux,  $J$ , of methane into the soil by Fick's law (as  $\text{cm}^3 \text{CH}_4$  per second ( $\text{s}^{-1}$ ) per unit cross-sectional area ( $\text{cm}^{-2}$ )):

$$J = D \cdot \frac{C_0 - 0}{z}$$

or, more generally we can write that at any point in the soil that the following condition must be satisfied:

$$J = D \cdot \frac{\Delta C}{\Delta z}$$

where  $\frac{\Delta C}{\Delta z}$  is the gradient in  $\text{CH}_4$  concentration (i.e., the change in concentration divided by the change in depth).

<sup>18</sup> In reality the system looks more like Figure 8.7, and actually, even more like Figure 8.8 ... adding considerable complexity (and dynamics).

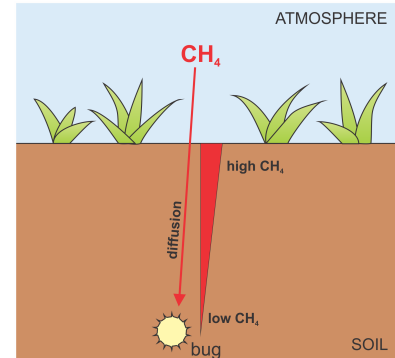


Figure 8.6: Idealized schematic of the soil- $\text{CH}_4$  system.

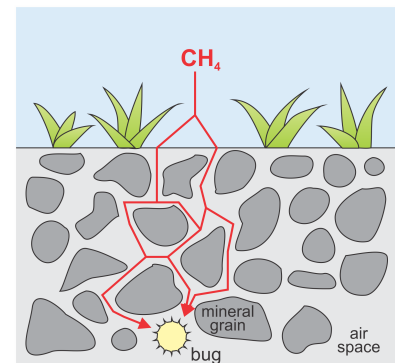


Figure 8.7: Slightly less idealized schematic of the soil- $\text{CH}_4$  system.

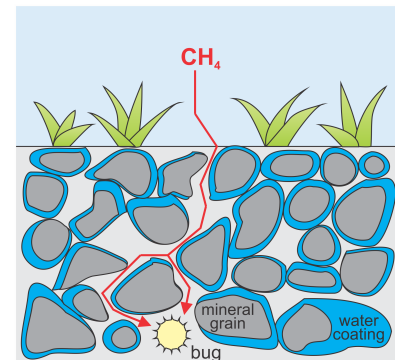


Figure 8.8: Even less idealized and almost realistic, schematic of the soil- $\text{CH}_4$  system.



If all there was to the soil methane system was consumption to zero at known depth, we could simply use an analytical solution to calculate the CH<sub>4</sub> flux into the soil. Unfortunately, life is rarely as kind, and there are a number of complications (see background material). For instance, the bugs do not all live at the same depth in the soil column (although that is the assumption made in *Ridgwell et al.* [1999]), nor have a constant activity throughout the year. Also, soil properties vary with depth, which affects the porosity and tortuosity of the soil (basically, how interconnected soil pore spaces are, and thus in effect how conductive the soil is to gaseous diffusion) and thus the diffusivity ( $D$ ) of CH<sub>4</sub> in the soil column, illustrated in Figure 8.7. We will assume an initial value for  $D$  of  $0.186 \text{ cm}^2 \text{ s}^{-1}$ .

Because we would quite like a general model for soil CH<sub>4</sub> uptake that was capable of accounting for these sorts of complications if necessary, we will solve the system numerically rather than restricting us to a simple analytical solution. This is what we will be doing in this exercise – constructing the basic model of atmospheric CH<sub>4</sub> diffusion into the soil, although there is not time in this exercise to go on and consider the metabolic consumption of atmospheric CH<sub>4</sub> by methanotrophic bacteria.

---

If we divide up the soil profile into 10 equally-spaced (equal thickness) layers<sup>19</sup>, the basics of the model will be an array with 10 rows, one (row) location in the array representing the CH<sub>4</sub> concentration in the pore space corresponding to each 1 cm thick interval of soil (see Figure 1). Thus, row #1 corresponds to the concentration in the 0-1 cm depth interval,  $C_1$ , #2 corresponds to the 0-1 cm depth interval,  $C_2$ , ... , and #10 corresponds to the 9-10 cm depth interval,  $C_{10}$ . We will also need to create an array to store the average depth,  $z_n$  at which each of the CH<sub>4</sub> concentrations is measured. These depths will be; 0.5 ( $z_1$ ), 1.5 ( $z_2$ ), 2.5 ( $z_3$ ), ... , and 9.5 cm ( $z_{10}$ ).

We are now ready to calculate the diffusion of CH<sub>4</sub> down the soil column. From the earlier equation, you know that you can relate the methane flux to the gradient in the soil, and the gradient between any two successive soil layers is equal to:

$$\frac{C_n - C_{n+1}}{z_{n+1} - z_n}$$

This is just to say, the difference between the concentration in any layer  $n$  and the concentration in the layer immediately below it (which will be number  $n + 1$ ) divided by the depth interval between the mid-points of the same two layers, which is the depth (from the surface) of the deeper layer ( $z_{n+1}$ ) minus the depth of the layer immediately above (which is layer  $n$ ).

Putting this all together, the downwards flux of CH<sub>4</sub> between

#### EXAMPLE OVERVIEW:

1. create function
2. create arrays and initialize model parameters
3. set up plotting (useful for later)
4. create time-stepping loop framework
5. add code to calculate fluxes:
  - (I): flux into surface layer
  - (II): flux into the (9) interior layers in a loop
6. add code to update concentrations based on fluxes:
  - (I): updating of first 9 layer concentrations in a loop
  - (II): updating of bottom-most layer

<sup>19</sup> It need not be 10 – choosing 10 layers of 1 cm thickness each, just simplifies things.

layers is given by:

$$J = D \cdot \frac{C_n - C_{n+1}}{z_{n+1} - z_n}$$

You can think of this system as analogous to the Great Lake model system<sup>20,21,22</sup> – there we had a series of reservoirs storing stuff (heavy metals), and there was a flow of material from one lake to the next. Here we have gaseous CH<sub>4</sub> in soil pore spaces rather than metals in solution in a lake, and we have diffusion of CH<sub>4</sub> from one soil level to another rather than a flow of water from one lake to another. The only real difference is that in the Lake Model more of the work was done for you and you were given the flow rates between lakes, whereas here you have to calculate the transport (diffusion) rate of CH<sub>4</sub>. The strategy for simulating the behavior of this system through time will be very similar though – stepping through time, and during each time step calculating the mass fluxes of CH<sub>4</sub> between layers and adding this to the pre-existing concentrations in each layer. The other difference with the Lake Model is that all the soil layers in an indexed array rather than being given different (lake) names, allowing you to use a loop.

OK – now for the to-do stuff ...

1. Create a new **m-file function**. Pass in the run length (in units of seconds) of the model simulation as a parameter, and e.g. call it `maxtime`. See the blurb from previously for how to define a function. If you want to be tidy: add a `close all` statement near the start of the *function*.<sup>23</sup>

2. Create a 10×1 vector array call `conc` and initialized with all zeros<sup>24</sup>. This is the variable array for storing the concentration of CH<sub>4</sub> in each 1 cm interval of the soil profile. Note that we are assuming no methane is present in the soil to start with (zero soil CH<sub>4</sub> concentrations is the initial condition of the model).

Also create a 10×1 vector array called `J`, again initialized with all zeros, to store the fluxes of CH<sub>4</sub> into each of the 10 soil layers from the one above (analogous to how you had the series of river fluxes associated with the various lakes in a previous exercise).

Then create a 10×1 vector array `z_mid` to store all the soil mid-layer depths (0.5, 1.5, 2.5, ... , 9.5). (This is a parameter array for helping in the plotting of soil CH<sub>4</sub> concentration against depth, later on.) Note that you need to create an array of 10 values, starting at 0.5, ending at 9.5, and with a step interval of 1.0. Go dust off the colon operator to create this vector array.

Also create a parameter (`conc_atm`) to store the concentration of CH<sub>4</sub> in the atmosphere. To keep things as simple as possible, you will be assuming units of cm<sup>3</sup> cm<sup>-3</sup>, so that the atmospheric

<sup>20</sup> Except less wet.

<sup>21</sup> And smaller.

<sup>22</sup> And in the soil ... OK, so not so much like the Great Lakes system ...

<sup>23</sup> Note that because the variables created in a function are *private* (and not seen by the rest of the MATLAB workspace), there is no need to issue a `clear all`. In fact: if you add a `clear all` at the start, you'll clear the (run length) variable that you have just passed in ... :(

<sup>24</sup> To save time – use the **MATLAB** function `zeros`.

CH<sub>4</sub> concentration becomes  $1.7 \times 10^{-6} \text{ cm}^3 \text{ CH}_4 \text{ cm}^{-3}$  (equivalent to 1.7 ppm), i.e.:

```
conc_atm = 1.7E-6;
```

Also, just for completeness, define a constant to store the depth at which the soil surface meets the atmosphere:

```
z_atm = 0.0;
```

Finally, define a parameter to store the value of the diffusivity constant  $D$  ( $0.186 \text{ cm}^2 \text{ s}^{-1}$ ):

```
D = 0.186;
```

3. Create a basic time stepping loop. Define a time-step length ( $\Delta t$ ) to take – this is the amount of time that going around the loop each time represents. Call the time-step length parameter  $\Delta t$  and assign it a value of 0.1 (s) (do this somewhere before the loop starts in the **m-file** but after the function definition line at the very top of the script). The model simulation length you want is given by the (passed) parameter `maxtime`, and each time around the loop lasts  $\Delta t$  in model time, so how many counts around the loop do you need to take ... ? If you call the loop counter `tstep`, then it should be obvious :o) that the start of the loop will look something like:

```
for tstep = 1:(maxtime/dt)
```

Yes? Before you do anything else, play with the function and check that the time-stepping loop is working and that you understand what it is doing. Try printing out (`disp()`<sup>25</sup>) the current loop value of `tstep` as well as the time elapsed in the model.<sup>26</sup> One way of displaying what is happening in the loop is to add a debug line such as:

```
disp(['time-step number = ' num2str(tstep) ', ...
time elapsed = ' num2str(tstep*dt) ' seconds']);
```

(All I am doing here is concatenating several strings together – a description of what is being written out followed by a value (a number variable converted to a string using `num2str`), then another description of what is being written out followed by a value, and finally the units of the second number.) If your function was called `ch4model` (for instance) and you type:

```
» ch4model(1.0)
```

you should now get something like:

```
time-step number = 1, time elapsed = 0.1
time-step number = 2, time elapsed = 0.2
time-step number = 3, time elapsed = 0.3
time-step number = 4, time elapsed = 0.4
time-step number = 5, time elapsed = 0.5
```

<sup>25</sup> The display line(s) should go inside the loop, of course.

<sup>26</sup> Equal to the loop count multiplied by the time-step length:

```
tstep*dt
```

```

time-step number = 6, time elapsed = 0.6
time-step number = 7, time elapsed = 0.7
time-step number = 8, time elapsed = 0.8
time-step number = 9, time elapsed = 0.9
time-step number = 10, time elapsed = 1

```

The loop has gone around 10 times because you asked for 1.0 s worth of model simulation (the passed parameter `maxtime`) and the time-step (`dt`) is defined as 0.1 s. Happy? (:o))

4. Run what you have so far and make sure that it works.<sup>27</sup>

Remember: build up a piece of computer code piece-by-piece, testing at each step before moving on. Believe me, there'll be more time for beers at the end compared to trying to write it all in one go and then not having a clue as to why it is not working ...

5. At the end of the function (i.e., after the loop has ended), plot the concentration profile of CH<sub>4</sub> in the soil column – you will want depth (cm) on the *y*-axis and concentration on the *x*-axis. Depth should run from 0 cm at the top to 10 cm at the bottom. Scale the *x*-axis so that concentration runs from 0 to  $2.0 \times 10^{-6}$  cm<sup>3</sup> cm<sup>-3</sup>. Also plot on the same graph as a point the atmospheric CH<sub>4</sub> concentration at the surface of the soil, whose value is held in the parameter `conc_atm`.<sup>28,29,30</sup>

6. Call the function from the command line and check again that everything is working OK. There should be no crashes (check for bugs and typos if not) and you should get a graph which has a vertical line running from almost the top (-0.5 cm) to almost the bottom (-9.5 cm) at a concentration of 0 cm<sup>3</sup> cm<sup>-3</sup>, together with a point at the top (depth = 0.0) marking the atmospheric CH<sub>4</sub> concentration of  $1.7 \times 10^{-6}$  cm<sup>3</sup> CH<sub>4</sub> cm<sup>-3</sup> (or 1.7 ppm if you have re-scaled the *x*-axis values). Check that you have this. Note that the CH<sub>4</sub> soil profile line can be hard to see because it runs along the axis. You can make the line thicker in the plot command by:

```
plot(conc(1:10), -z_mid(1:10), 'LineWidth', 3);
```

You can also fill in the atmospheric CH<sub>4</sub> point by passing the optional parameter `filled` to the `scatter` function..

7. So far this is not exactly very exciting (\*yawn\*). In effect, you have a model for a soil system in which the soil is capped by an impermeable layer at the surface (preventing any entry of atmospheric CH<sub>4</sub> into the soil) and nothing happens.

8. So now get model actually calculating something. Within the time-stepping loop you are going to calculate the flux of CH<sub>4</sub> between each layer. The concentration units of CH<sub>4</sub> are cm<sup>3</sup> CH<sub>4</sub> cm<sup>-3</sup>. The length scale is cm. The diffusivity of CH<sub>4</sub>, *D* has units

<sup>27</sup> Note that because the variables in a MATLAB function are private (and are thus not listed in the Workspace window), if you want to check the values in this array you could first leave off the semi-colon from the end of the line so that **MATLAB** prints the array contents to the screen. Or, explicitly add in a `disp()` line. Or ... add a breakpoint somewhere in the code and view the variable values when the program pauses.

<sup>28</sup> `hold on` and then using the `scatter` function is probably the easiest way.

<sup>29</sup> Note that **MATLAB** does not like you trying to plot the *y*-axis with the numbers getting more negative as you go up the axis. One way around this is to plot the negative of the depth on the *y*-axis; e.g.:

```
plot(conc(1:10), -z_mid(1:10));
axis([0 2.0E-6 -10 0]);
```

so you really have the *y*-axis scale going from 0 cm at the top, to minus 10 cm at the bottom. (If you are clever, there are ways around this involving explicitly specifying the labeling of the *y*-axis ...)

<sup>30</sup> Also note that if you want your concentration scale in more friendly units, such as ppm, then you will need to scale the values you are plotting to make them 10<sup>6</sup> times bigger; i.e.:

```
plot(1.0E6*conc(1:10), -z_mid(1:10));
axis([0 2.0 -10 0]);
```

of  $\text{cm}^2 \text{s}^{-1}$ . So if we apply dimensionality analysis (basically, just working out the net units) we get:

$$J = \text{cm}^{-2} \times \text{cm}^3 \text{CH}_4 \text{cm}^{-3}/\text{cm}$$

which comes out to give  $J$  in units of  $\text{cm CH}_4 \text{s}^{-1}$ ! This looks a bit screwed up. However, what area of soil (the cross-section of the column) is the diffusion occurring across? The vertical length-scale of the 1D model has been defined, but what about whether the soil column is a nano-meter across or the area of the whole Earth? Assume that the cross sectional area of the 1D model is  $1 \text{ cm}^2$  (i.e.,  $1 \text{ cm} \times 1 \text{ cm}$ ). Therefore, the flux of  $\text{CH}_4$  is occurring in a  $1 \text{ cm}^2$  unit cross sectional area model, with units of:

$$J = \text{cm}^{-2} \times \text{cm}^3 \text{CH}_4 \text{cm}^{-3}/\text{cm} \times \text{cm}^2$$

or  $\text{cm}^3 \text{CH}_4 \text{s}^{-1}$ . This is much more reasonable (and  $\text{cm}^3$  of  $\text{CH}_4$  can easily be converted into units of moles or g of  $\text{CH}_4$  if you needed to).

9. Before adding in the meat of the model (the calculation the fluxes of  $\text{CH}_4$  between the pairs of  $1 \text{ cm}$  layers in the soil column), it is easiest to calculate separately the special case of the flux from the atmosphere into the first layer. The average distance ( $\Delta z$ ) over which diffusion occurs is only  $0.5 \text{ cm}$  in this case (measuring from the surface (zero height) to mid-depth of the first  $1 \text{ cm}$  thick layer). Referring to the equations previously, but recognizing that the  $n = 0$  layer doesn't exist because it is the atmosphere<sup>31</sup> (so `conc(0)` and `z_mid(0)` have been replaced by `conc_atm` and `z_atm`, respectively) you should see that the flux of  $\text{CH}_4$  into the first soil layer from above is:

$$J(1) = D * (\text{conc\_atm} - \text{conc}(1)) / (\text{z\_mid}(1) - \text{z\_atm});$$

10. Now for the main course of your modelling feast. It should be obvious(!) that what happens for layers 2 through 10 is basically identical – i.e., for each of the layers  $n = 2$  through  $n = 10$ , the flux of  $\text{CH}_4$  into layer  $n$  from the layer above ( $n - 1$ ) can be written:

$$J(n) = D * (\text{conc}(n-1) - \text{conc}(n)) / (\text{z\_mid}(n) - \text{z\_mid}(n-1));$$

So, you could write a little loop, going from  $n = 2:10$ , and calculate the value of  $J(n)$  within the loop.<sup>32</sup>

11. Make sure that you are happy with what you have done so far. You have calculated the  $\text{CH}_4$  flux from the atmosphere into the first soil layer ( $n = 1$ ). You have done this on its own because it is a special case – there is no soil layer immediately above, only the atmosphere. Then you have calculated the fluxes into each soil layer ( $n$  from 2 to 10) from the layer above within an  $n$  loop (because it is easier than writing out the same equation 9 times!).

<sup>31</sup> And also because you cannot start indexing a vector in **MATLAB** at zero.

<sup>32</sup> Don't forget that you have just calculated the first  $n = 1$  layer flux ( $J(1)$ ) already.

Although you are not yet updating the concentration of CH<sub>4</sub> in the soil layers, it is worth running the model again to check that that all the new things that have been added to the model work. Do this, and check that you can still call the function without **MATLAB** errors appearing (although this does not guarantee that you have not made a mistake ...).

12. So, all that is left to do now is to update the concentration of CH<sub>4</sub> in each soil layer and see what happens ... To keep it simple, assume that the soil has a porosity of 1 cm<sup>3</sup> cm<sup>-3</sup> (i.e., all air space and no actual soil!!!) – see *Ridgwell et al.* [1999] to get a feel for how complicated gas diffusion in a real soil becomes and how you must modify the diffusion coefficient to take into account different factors (such as soil type and moisture content). To update the CH<sub>4</sub> concentration in soil layer  $n$  due to the flux of CH<sub>4</sub> from above (layer  $n - 1$ ) you must add a volume of CH<sub>4</sub>, given by the calculated  $J_n$  value (in cm<sup>3</sup> of CH<sub>4</sub> per second) multiplied by the time-step interval (in s). You must also take into account the loss of CH<sub>4</sub> from each soil layer  $n$  as CH<sub>4</sub> diffuses into the layer below ( $n + 1$ ). So, just like you calculated the new metal pollution concentrations in the lakes by taking account what was there to start with, plus any gain, minus any losses, the concentration change for layer  $n = 1$  for instance (but don't write this in), is simply;

$$\text{conc}(1) = \text{conc}(1) + \text{dt} * J(1) - \text{dt} * J(2);$$

This is saying that the new CH<sub>4</sub> concentration in layer  $n = 1$  is equal to the concentration at the previous time-step, plus the CH<sub>4</sub> that diffuses into the later from above ( $J(1)$ ), minus the CH<sub>4</sub> that diffuses out of the layer at the bottom ( $J(2)$ ). Does this make sense? You need to exercise your paw if not.

13. You could write out 10 equations to update the 10 soil layer CH<sub>4</sub> concentrations, or ... use another loop! You will have to be careful, because when you get to layer  $n = 10$ , there is no flux downwards because it is the bottom of the model. The bottom boundary condition of the model is then that there is no downwards flux. (We could have defined the soil column to be deeper than this, but it is always better to keep any model you are constructing as simple as possible to start with.) You will therefore have to treat the bottom-most ( $n = 10$ ) layer separately, but you can still loop through from  $n = 1$  to 9, and use the same equation. So, create a new loop, just after the  $n=2:10$  one, and set its counter (you can re-use the name  $n$ ) going from  $n=1:9$ . Within this second  $n$  loop, update the CH<sub>4</sub> concentrations for layers  $n = 1$  through 9:

$$\text{conc}(n) = \text{conc}(n) + \text{dt} * J(n) - \text{dt} * J(n+1);$$

Now add in the code to update the  $n = 10$  layer  $\text{CH}_4$  concentration (i.e., adding just the flux from above ( $J(10)$ ) to the current  $\text{conc}(10)$  concentration value).

Now you are done. Hopefully. The overall structure of loops and things should now look something like (NOTE: not necessarily exactly like):

```
function ...
% (1) initialize model variables and set model parameters
...
%
% (2) start of time-stepping loop
for tstep = 1:(maxtime/dt),
    % calculate the CH4 flux from the atmosphere into
    the first
    % soil layer
    J(1) = ...
    % calculate the CH4 fluxes from one soil layer to
    the next
    % (n=2:10)
    for n = 2:10
        J(n) = ...
    end
    % update the concentration of CH4 in each of the
    soil layers
    % (n=1:9)
    for n = 1:9
        conc(n) = ...
    end
    % and finally update the concentration for the
    special case
    % of n=10
    conc(10) = ...
end
% (end of time-stepping loop)
%
% (3) plot results
...
end
```

Run it for 10s (`»ch4model(10.0)`) and see. You should see a profile of decreasing  $\text{CH}_4$  concentrations as you go down deeper into the soil, looking something like Figure 8.9.

Now try a longer model run (100 s) (`»ch4model(100.0)`) and see what happens. You should get something like Figure 8.10.

Go find out when the system (approximately) reaches equilibrium (i.e., the profile stops changing with time). You will need to judge when any further changes are so small they could not possibly really matter.

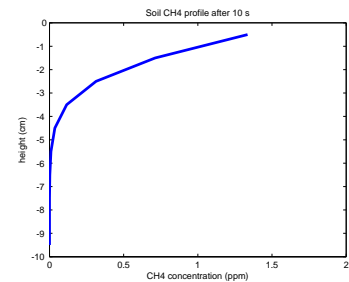


Figure 8.9: Soil profile of  $\text{CH}_4$  after 10.0s of simulation.

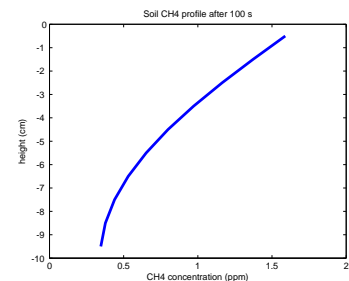


Figure 8.10: Soil profile of  $\text{CH}_4$  after 100.0s of simulation.

KEEPING WITH THE SAME EXAMPLE<sup>33</sup> and having constructed the basic diffusion framework for the model, we can explore what happens if consumption of CH<sub>4</sub> (by methanotrophs) occurs within the soil (as well as exploring the numerical stability and hence choice of time-step duration and grid resolution, of the model).

First, take the `ch4model` (or whatever named) function and add a second input parameter to set the time-step length. You should then have two input parameters (`maxtime` and `dt`).<sup>34</sup> By calling the function from the command line, with a model simulation duration of 100 s, play around with the time-step length. Approximately, what is the longest time-step you can take before the model becomes numerically unstable? What are the characteristics of the soil CH<sub>4</sub> profile that lead you to suspect instability occurring in the numerical solution? The onset of instability might look something like Figure 8.11.

Now ... it just so happens that some top profs (me!?) have told you that there are some bugs – methanotrophs (see *Ridgwell et al. [1999]*) that live deep down in the soil. From this, you assume that they will be present only in the deepest ( $n = 10$ ) soil layer in the model. They just sit there, munching away on CH<sub>4</sub> that diffuses down from the atmosphere into the soil pore-space. A bit like idle grad students living on a diet of pizzas.<sup>35</sup> The bugs consume the CH<sub>4</sub> present in the soil pore space at a rate that is proportional to the concentration of CH<sub>4</sub> in the soil (makes sense – the more CH<sub>4</sub> food source there is to metabolize, the more than they will remove per second). Call this rate constant e.g. `munch_rate`. It has units of fractional removal per second. In other words, if the concentration of CH<sub>4</sub> in layer  $n = 10$  is `conc(10)`, then in one second:

$$\text{munch\_rate} * \text{conc}(10)$$

$\text{cm}^3 \text{CH}_4 \text{cm}^{-1}$  will be lost from the soil pore space. So, if you had a rate constant (`munch_rate`) of  $0.5 \text{ s}^{-1}$ , then each second, half of the CH<sub>4</sub> in layer  $n = 10$  would be removed. Of course, the time-step in the loop might not be 1.0s – if you had `dt=0.1`, for instance, then the loss of CH<sub>4</sub> each time around the loop would be:

$$0.1 * \text{munch\_rate} * \text{conc}(10)$$

$\text{cm}^3 \text{CH}_4 \text{cm}^{-1}$ . Are you following so far ... ?

Now, add a third parameter that is passed into the soil CH<sub>4</sub> model function for the rate constant. Modify your equation for the updating of the CH<sub>4</sub> concentration in the deepest ( $n=10$ ) soil layer to reflect the presence of the methanotrophs. Call the soil CH<sub>4</sub> model function; pass a time-step of 0.1 s and a methanotroph CH<sub>4</sub> consumption rate

<sup>33</sup> OVERVIEW:

1. adapt model and explore choice of time-step
2. adapt model and explore choice of layer thickness / number of soil layers
3. add methanotrophs (CH<sub>4</sub> sinks)
4. play!

<sup>34</sup> Note that you will have to comment out (or delete) the line in the code where previously you defined the time-step length as fixed with a value of 0.1 s.

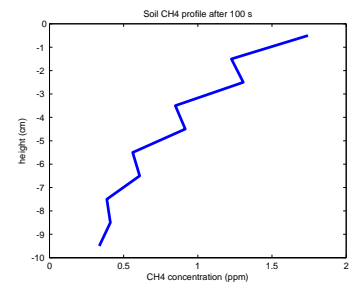


Figure 8.11: Soil profile of CH<sub>4</sub> after 100.0s of simulation with an extremely marginal choice of time-step length.

<sup>35</sup> Except students mostly don't live in the cold damp dirty ground.



constant of  $1.0 \text{ s}^{-1}$ . Your function call should look something like this at the command line;

```
» ch4model(xxx, 0.1, 1.0)
```

where `xxx` is the duration of the simulation<sup>36,37</sup>. How many seconds (approximately) does it take for an equilibrium profile to be established (i.e., what was the simulation duration that you used to create your plot?). What, ultimately, is the shape of the soil profile of  $\text{CH}_4$  concentration, and why?

Now ... lets say that you then go out into the field and take samples from each 1 cm thick interval of a 10 cm soil profile. You incubate the soil samples in sealed flasks with  $\text{CH}_4$  initially present in the headspace (a fancy word for the air or gas above a sample in a container). Hey – you observe that  $\text{CH}_4$  is removed in all flasks, equally. Someone screwed up(!) – these bugs live throughout the soil column, not just at the bottom. You'd better update your model in light of these new scientific findings.

Add a term (within the 2nd  $n$  loop in which you update the  $\text{CH}_4$  concentrations) to reflect the consumption of  $\text{CH}_4$  in the layers  $n = 1$  through 9. (You can keep the term for consumption in the  $n = 10$  layer.) Since the bugs are spread out through 10 layers rather than being concentrated in one (at the bottom), presumably the consumption rate is only  $1/10$  of your previous rate value. So use `munch_rate = 0.1` (i.e., a rate constant of  $0.1 \text{ s}^{-1}$ , rather than the value of  $1.0 \text{ s}^{-1}$  that you used before) for all subsequent calculations. Call the soil  $\text{CH}_4$  model function with a time-step length of 0.1 s and determine the steady state soil (equilibrium)  $\text{CH}_4$  profile (Figure 8.13). What shape does this remind you of ... and why?<sup>38</sup>

A couple of slightly more challenging modifications to try now:

1. Alter the model so that you can also pass into the function, the number of soil layers that are represented in the upper 10 cm – equivalent to altering the thickness of each layer. This change is a little more involved than simply altering the time-step duration. For instance, now, rather than  $n$  (the number of layers) going from 1 to 10, they are now counted from 1 to  $n_{max}$ <sup>39</sup> (the number of model layers you pass into the function)
  2. Add in a parameter controlling the maximum depth of the soil column represented (replacing the fixed 10 cm assumption from previously).
  3. Try adding a source of  $\text{CH}_4$  at the base of the soil column.<sup>40</sup>
- <sup>41</sup> Units should be:  $\text{cm}^3 \text{ CH}_4 \text{ cm}^{-3} \text{ s}^{-1}$ . But now much (i.e. what rate of methane production would be reasonable)? You could play about, trying different values until finding one that did not

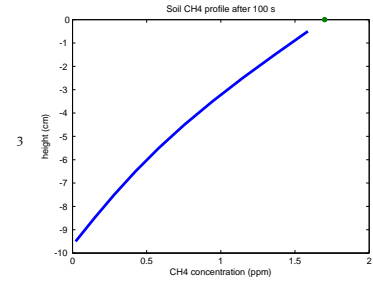


Figure 8.12: Soil profile of  $\text{CH}_4$  after 100.0s of simulation, with  $\text{CH}_4$  uptake at the base of the profile with a rate constant of  $1.0$  per s.

<sup>38</sup> There is in fact an analytical solution to this profile – can you derive it?

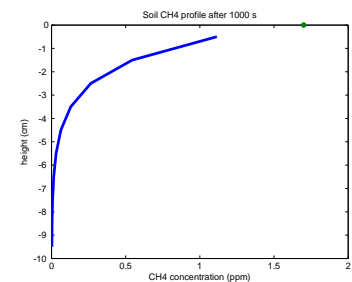


Figure 8.13: Equilibrium soil profile of  $\text{CH}_4$ , with  $\text{CH}_4$  uptake throughout the soil column with a rate constant of  $0.1$  per s.

<sup>39</sup> For which you might call the variable, e.g. `n_max`.

<sup>40</sup> This is quite physically plausible and might reflect (in order of decreasing likelihood): a water-logged, anoxic layer at depth, thawing permafrost, or a natural gas seep.

<sup>41</sup> Note that now you have 2 different boundary conditions in the model – a fixed concentration in the atmosphere at the surface, and a fixed flux at depth.

produce anything insane. Not a very satisfying approach. You could certainly look up in the literature measured soil production values (a much better approach). You could also get a feel for a possible order-of-magnitude by contrasting with the previous consumption flux (from the atmosphere). Actually, you have not looked at this so far (the total atmospheric CH<sub>4</sub> consumption flux) and maybe should have as it is what matters in terms of the soil being an effective sink, or not, for atmospheric CH<sub>4</sub>. To do this – you need to extract from the model, the CH<sub>4</sub> flux from the atmosphere into the first soil layer (why?). Do this and make it the returned values from the function. Now set the production (at depth) rate similar to the net (from atmosphere) consumption flux from before (with methanotrophic activity throughout the soil profile). You should obtain a profile (at steady state) that is approximately symmetrical in depth<sup>42</sup> – e.g. Figure 8.13.

4. Finally ... there should be (there is!) a value for the production rate at depth, at which the flux into the atmosphere is zero. (There are certainly some very large production rates at depth for which the flux from the atmosphere is negative, i.e. there are net emissions of CH<sub>4</sub> \*to\* the atmosphere. Can you find this value (which makes the net exchange zero) ... \*without\* trial-and-error?<sup>43</sup>

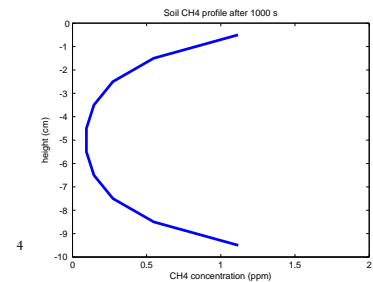


Figure 8.14: Example equilibrium soil profile of CH<sub>4</sub> with production at depth.

<sup>43</sup> Your function returns the net flux and you need to search for the production rate value that minimizes this net flux. Meaning you need to construct a search algorithm, testing a larger production rate of the net flux is positive, and a smaller value if the net flux it is negative.

# 9

## *Graphical User Interfaces (GUI)*

In this chapter we'll learn how **MATLAB** can create a simple *Graphical User Interface (GUI)*, which you can use to interface to your program with (as an alternative to e.g. the command line). Scientifically ... this is not particularly useful, although it is fun(ish) and is how all computer/device software/apps tend to 'work' these days (excepting devices (e.g. wifi routers) that use a web-browser for their interface, but then that is effectively a *GUI* within a *GUI* ...).

## 9.1 MATLAB GUI basics

**MATLAB** kindly<sup>1</sup> 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 9.1 should appear. We’ll only concern ourselves with the default option amongst the (4) ‘GUIDE templates’ – ‘Blank GUI (default)’<sup>2</sup>. As for the tick-box ‘Save new figure as:’ – we’ll leave this alone<sup>3</sup>. 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*).

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 window (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 note it has:

- Close/minimize/maximize buttons at the top right (plus the window can be re-sized by grabbing the corner and dragging the mouse).
- A title at the top (‘title bar’) with a cute (\*barf\*) **MATLAB** icon.
- 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 ‘Create Existing GUI’ tab, there is:
  - A list (of GUIDE 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 button labelled ‘Browse’.
- (In the ‘Open Existing GUI’ tab ... nothing much (yet) going on.)

<sup>1</sup> For once, it is not a sperate, zillion-dollar license ...

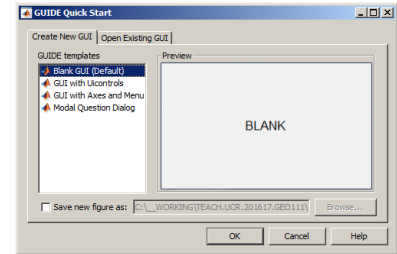


Figure 9.1: Starting GUI window of the **MATLAB GUIDE**, GUI design tool.

<sup>2</sup> So don’t go randomly clicking on anything just yet!

<sup>3</sup> 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.)

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<sup>4</sup>. 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 in the initial **GUIDE** window – do it now.)

Rather than creating a blank m-file and/or some basic code first<sup>5</sup>, **MATLAB** throws you straight into a window design tool as per Figure 9.2. There is a lot going on here, but start by noting that 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 icons 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?) – Current Point and Position. To the left of the window is a group of icons<sup>6</sup> 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. (At least, that is what I see!)

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<sup>7</sup> you'll have to do. So we'll start as simply as possible and build up.

<sup>4</sup> That is: 1980s, as much as some might believe **Microsoft** has made little progress since the 1880s ...

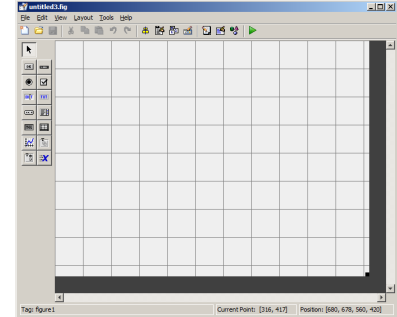


Figure 9.2: (Blank) GUI window editor GUI window.

<sup>5</sup> 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.

<sup>6</sup> Still no re-appearance of the **MATLAB** icon!

<sup>7</sup> Which has a steep power relationship with the amount of code.

### 9.1.1 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, which should be already open if you haven't fatally mucked about with it (or open up a new *GUI* by typing `guide` (lowercase) at the command line again) – identify the Static Text icon (by hovering the mouse pointed over an icon, its function is revealed). Click (left mouse button) on it. The mouse pointer, when over the gridded design pane, should change to a cross-hairs mouse pointer icon.<sup>8</sup> 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 with a normal **Windows** (or **MacOS** etc.) window.

So far, the (static) text object has 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<sup>9</sup>. Whoa! That's a long list of ... actually, *properties* of the *object* (thats two new buzz-words in one – *object* and *properties*). 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.<sup>10</sup> 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, why not also play with some of the other properties if you like (at least, the ones that you can make a reasonably informed guess as to what they do). Maybe you end up with a design window looking like Figure 9.3. Note that the effect of your changes is only shown if you e.g. hit Enter or click on a different property. If you accidentally click outside of the text object and 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.)<sup>11</sup>

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 window and there is a whole bunch of code in it (a series of

<sup>8</sup> Note that this is to facilitate the positioning of the icon rather than being anything about guns and shooting at the coders behind **Windows**.

<sup>9</sup> 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 are going to create in your *GUI*(s).

<sup>10</sup> In reality: **MATLAB** is secretly writing the relevant code and setting the parameter value ...

<sup>11</sup> 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.

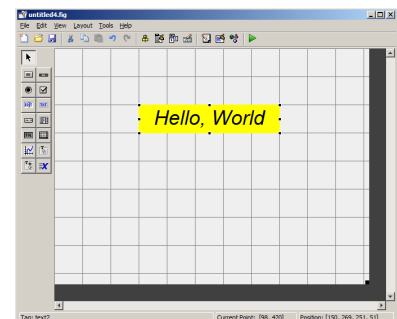


Figure 9.3: Design of the Hello, World window!

*functions* in fact). Note that the code file has a filename the same as you entered in for the .fig window but now with a .m extension (and so is presumably directly associated with the figure you just created). There is nothing we need worry about ... yet. In fact, half the file is taken up with a main *function* that has the comment: DO NOT EDIT. Please take this advice ... :o)

In fact, you get given the framework code for 3 functions:

1. The long one at the top, starting/ending:

```
function varargout = FUNCTIONNAME(varargin)
% FUNCTIONNAME MATLAB code for FUNCTIONNAME.fig
...
...
...
% End initialization code - DO NOT EDIT
```

(where FUNCTIONNAME is the name of your function, i.e. the filename you saved the GUI as).

This code is automatically generated and defines the *function* for your app/program.

Do not edit this code, and do not place any of you own code within it. IGNORE IT!!!

2. The middle function, starting/ending:

```
% -- Executes just before FUNCTIONNAME is made visible.
function FUNCTIONNAME_OpeningFcn(hObject, eventdata,
handles, varargin)
% This function has no output args, see OutputFcn.
...
...
% UIWAIT makes FUNCTIONNAME wait for user response (see
UIRESUME)
% uiwait(handles.figure1);
```

Note that **MATLAB** did not generate an 'end' to formally define the end of the *function* ... assume that the last comment line is the end of this function.

Do not edit this code, and do not place any of you own code within it (for now).

This *function* allows you to execute any code before the window appears. Such code is typically associated with initialization (setting sup arrays and defining parameters etc.)

3. The third and final one, starting/ending:

```
% -- Outputs from this function are returned to the
command line.
function varargout = FUNCTIONNAME_OutputFcn(hObject,
eventdata, handles)
% varargout cell array for returning output args (see
VARARGOUT);
...
...
% Get default command line output from handles structure
varargout{1} = handles.output;
```

simply allows you to set any output (*function* return) variables that you wish to pass back to the command line. (You need not pass anything back.)

Again – do not edit this function. Assume that the last line of the function code is `varargout{1} = handles.output;` and that really, MATLAB should have added an 'end' to formally define the end of the *function*.

Although there looks like a lot of stuff here, the code is automatically generated and generic and there are both a bunch of blank lines that bloat it all, and lots of comment lines, mostly briefly describing the functions and their inputs. (Few of these bit of information we care about.)

Close the design window (and the code editor if it distracts you). At the command line, type the filename of your GUI *function* (i.e. the **m-file** name – but as per with all your scripts and function – without the .m extension) to run the automatically generated code **m-file**. A window opens up ... the 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*).

### 9.1.2 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 of your own ... so we'll start with the simplest possible action – a *button* that performs a simple action (closes the window).

Re-run the `guide` program 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 9.4. 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 then save it (using a different filename to previously).

When it saves, **MATLAB** again opens up the code associated with the figure window that it has automatically generated. There is slightly more code in the file this time and shortly, you'll need to look at it. But right now: ignore it, and type the name of your **m-file** file at the command line. Again, you'll get a window opening with the push button you created in it. Click on it. It does seem

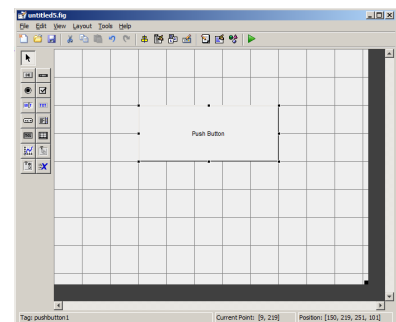


Figure 9.4: Design window with a default push button object.



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** (`» 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.<sup>12</sup> By default, the name is `pushbutton1`. Edit this to ... `goawayButton` (or pick an alternative name) and re-save the GUI.

<sup>12</sup> 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 as the `.fig` figure, remember). In the file we have:

- The main function which we can ignore (and indeed apparently should not be edited!). But for completeness, it consists of:

- The *function* definition header line:

```
function varargout = NAME(varargin)
```

where `NAME` is the name you assigned the file.

- Some comment lines:

```
% NAME MATLAB code for NAME.fig
% NAME, by itself, creates a new NAME or raises the
existing
% singleton*.
etc etc ...
```

Note that there is a continuous block of comment (%) lines.

**MATLAB** treats this as the help text on function `NAME`.

- Then some more, but separated (by blank lines) comment lines.

- Then the body of the function, starting with:

```
% Begin initialization code - DO NOT EDIT
```

and then ending with:

```
% End initialization code - DO NOT EDIT
```

- `function` `NAME_OpeningFcn` which is executed when the GUI is started up. This is the place to put code for initializing models or whatever (hence the automatically generated part of the *function* name – `OpeningFcn`).

It is not obvious (to me) what either:

```
% Choose default command line output for NAME
handles.output = hObject;
```

or

```
% Update handles structure
guidata(hObject, handles);
```

```
270 str = 'do you like bananas?'
```

actually do ... so ignore these lines for now.

If you need to execute any code when the program/app first runs, you will place it after these lines.

- There follows another *function* call:

```
% -- Executes just before NAME is made visible.)  
function NAME_OpeningFcn(hObject, eventdata, handles,  
    varargin)
```

which seems to prepare any data that you wish to return from the main function and back to the command line.

Textbooks helpfully say to ignore this. Great idea.

- Finally, there is:

```
function goawayButton_Callback(hObject, eventdata, handles)
```

This *function* is executed when your 'Go Away!' push button is pressed. Anything you wish to 'happen', in terms of code executed, when you click on this particular button, goes here in this function.

Note that **MATLAB** does not formally end any of the functions with `end`. Don't get confused as to where to place code – assume that when the next *function* starts when the next `function` definition line appears (or the comment line immediately preceding it) and that the previous function has then now ended. If it helps – add in lines (with `end`) to end each *function*. Or perhaps add some ASCII art/comment line before each new function to help make it clearer, e.g.

```
% === END FUNCTION =====
```

In this simple *GUI*, we have only one figure and it is *active* (it has the mouses' attention)<sup>13</sup>, so we could simply use the `close` command ('deletes the current figure') on its own (just this on one line). Insert this simple command (`close`) in the

```
function goawayButton_Callback
```

*function* – after the *function* definition line and the comment lines that immediately follow it.

Save the **m-file** and run it. Now, when you click on the 'Go Away!' push button, the window does indeed 'go away' (aka, it closes).

Phew! So, to recap – you have created a program with a window, and within that window a single Push Button object. In the design window, you gave that button a special property, such that when clicked, a message (an *event*) would be passed back to your program. The code (a *function*) that responds (is called) when the button is clicked was automatically generated for you, but with no code inside. You added the code to close the program/window in response to the mouse click.

<sup>13</sup> Often in operating systems – the active window, i.e. the one that is the one to respond to mouse clicks or key presses, has its titlebar highlighted in a bright color (while inactive ones might be grey.)

### 9.1.3 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. Ideally – sketch out a design on paper<sup>14</sup> first.

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 ... but 2 of them rather than just 1).
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, leads to a different message being displayed.

The latter part is not as bad as it sounds. We could have no text initially in the 2nd (static) *text box*. We just need to change its text property (i.e. change the initial no text, to the text of our message). This is mostly a case of using the unique identifier of this text box object AND the identifier of the text property (of the text box object). i.e. you need two bits of information – the ID of the text box, and the ID of the property of the box that sets the actual text to be displayed. You'll see how this pans out shortly.

OK ...

Firstly – re-run **GUIDE** (`>> 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 that 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 9.5 and certainly not with those color choices ... but you get the idea.

For each of the objects (2 text boxes and 2 push buttons), rename them (i.e. edit the Tag property) to something memorable.

The code that **MATLAB** generates for `bananas.m` (my name choice! it need not be yours ...) 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 (so a total of 2 'useful' *functions* – one for each button object, in addition to the 3 ones **MATLAB** generates and you are ignoring). The end of the automatically generated **MATLAB** code now looks like:

<sup>14</sup> The white flat stuff that you write on. Maybe you have forgotten what it is? Clue: it is not an app on iTunes.

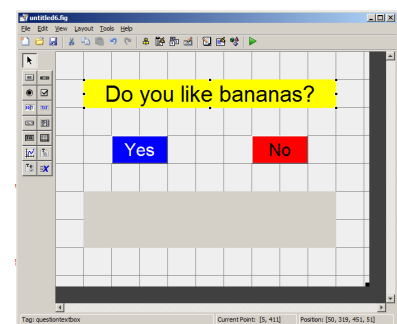


Figure 9.5: (completely) Bananas design window.

272 str = 'do you like bananas?'

```
% -- Executes on button press in yesbutton.  
function yesbutton_Callback(hObject, eventdata, handles)  
...  
...  
% -- Executes on button press in nobutton.  
function nobutton_Callback(hObject, eventdata, handles)  
...  
...
```

(here I have assigned to the two buttons in the property editor window, tags of: `yesbutton` and `nobutton` so as not to get confused as to which button *function* corresponds to which button object in the window).

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** does get all weird here.<sup>15</sup> 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. Think of it as rather than storing information in a database about Luna and by name, information might be stored by SSN instead. So to retrieve or write information about Luna, you do it via their SSN.<sup>16</sup> Here, we want to change a property (the displayed text of the 2nd text box), and you are going to have to get its ID first in order to do it.

[Note that all the code you write is going to in one or other of the Callback functions. There will be a similar set of lines of code in each.]

First off, you can get the ID of the object property using the `findobj` *function* and assign the result to some memorably variable<sup>17</sup>, 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 – if you chose a different name, or did not change the default name, you will need to adjust this line of code to reflect that).<sup>18</sup> You could put this line of code at the start (after the *function* definition, and after the initial comment lines) of the *function* `yesbutton_Callback` and because you have two buttons, both of which will need to be able to change the text in the 2nd text box – you place the same line of code also in the *function* `nobutton_Callback`

<sup>15</sup> Actually, no weirder than **netCDF**. Or arguably, **Python** ...

<sup>16</sup> Or you could think about university student databases and access via the unique student number.

<sup>17</sup> Here, the `h` bit stands for 'handle' but you might chose `id` for ID instead?.

<sup>18</sup> 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*.

Now that we have the ID of the 2nd text box (assigned to the variable `h_answertext`), we can set its property (from containing no text at all, to displaying 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, we obtained the handle of the text box object and assigned it 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, which goes after you have assigned the test box handle to `h_answertext`, is:

```
set(h_answertext,'String','Yes, it is an excellent fruit.');
```

The complete(!) code in `function yesbutton_Callback` should then look like:

```
h_answertext = findobj('Tag','answertextbox');
set(h_answertext,'String','Yes, it is an excellent fruit.');
```

Save and run the program. You could see something like the result shown in Figure 9.6 (if you click on Yes).

Now extend your program so that an alternative answer is provided if the No button is instead clicked. This is going to pretty much well be a duplication of the code you have already written for the Yes button.

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`.<sup>19</sup> Try this – i.e. obtain the handle of the Yes button (and e.g. assign to to the variable `h_button`) within the `function yesbutton_Callback function` code, and set a (different) color.

---

To put your coding skills more 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 one). You'll need a variable to remember whether any button has been pressed and assign this an initial value of `false`, e.g.

```
var_pressed = false;
```

The idea is whenever either button is pressed, `var_pressed` will become (will be set to) `true`. So before displaying the message in either 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

```
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.
```

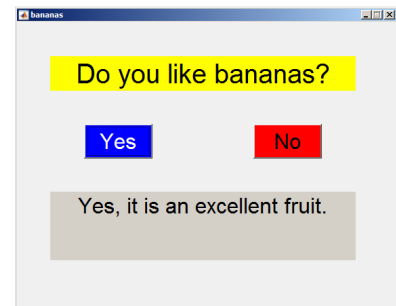


Figure 9.6: (completely) Bananas GUI in action.

<sup>19</sup> For example:

```
set(h_answertext,'BackgroundColor','g');
```

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 ... else ...` in each of the 2 callback *functions* to display different messages depending on the value of var\_pressed, and a single line of code that initializes the value of var\_pressed to *false* – this goes in the initialization *function* that you have been ignoring up to now, called `function bananas_OpeningFcn` (assuming your GUI *function* name is `bananas`).

There is just one problem ...

*Variables in functions* are 'secret' (*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 actually 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 (after the function definition line) 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*.

What is going on is that in each *function* that a *variable* is defined as being *global* – the value of that *variable* is linked, so that any change made to the value of that *variable* in any *function*, is seen by all the other ones.<sup>20</sup>

The complete code for the Yes button call back *function* (the No button code is almost identical) 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;
```

<sup>20</sup> Like multi-way **Skype** call, where the global definition is each person connecting to the conference call.

and with the code:

```
global var_pressed;
var_pressed = false;
```

appearing in `function` `bananas_OpeningFcn` (to initialize the value of `var_pressed` to `false`).

#### 9.1.4 Simple GUI responses [Sliders]

We can create a variant on the previous program to illustrate numerical input and output, and introduce the *Slider* object.

1. First create a new (blank) *GUI*.
2. Add a *Static Text* box *object* to ask: 'On a scale of 0 to 10, how much do you like bananas?' (replace the default text by editing the properties of the *object*).
3. Add a second *Static Text* box *object* to report the value. Create it blank to start with (i.e. delete the default text).
4. Add ... a *Slider*! Double-click to edit the *Slider* object. Firstly, note that there is a *Max* and *Min* property – these are the values assigned when the *Slider* is at its maximum and minimum position, respectively. Since you want a score for 0 to 10 – edit the *Max* value. There is also a *Value* property which will be the default value of the *Slider* when the program/app starts up. (If you want change the default value.) Make any other e.g. cosmetic changes you fancy. Close the editor ('Inspector') window.
5. Save the *GUI*.

In place of the button *Callback functions* in the automatically generated **MATLAB** code, you now have:

```
% -- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)
```

(although if you were paying attention earlier, you would have named the *Slider* something helpful rather than the default of `slider1`).

And a last *function* (`slider1_CreateFcn`) that we shall ignore.

This is not so different from the button example – when the slider bar is dragged, or the up/down arrows are clicked (and the slider bar moved that way), this *function* is called. It is then up to you (in code) to:

1. Read the value of the *Slider*.
2. Do something with that value (i.e. display it)

```
276 str = 'do you like bananas?'
```

Also as before, we need to get the ID of the Slider object, and then read its Value property.

```
h_slider1 = findobj('Tag','slider1');  
bananaindex = get(h,'Value');
```

(which goes in the `slider1_Callback` function).

To set the text in the Static Text box object, as before you need to obtain the ID of the object:

```
h_text2 = findobj('Tag','text2');
```

(here again ... not the greatest of *variable* name choices ...)

Now simply set the String property of the Static Text box, to the value of the Slider, contained in variable `bananaindex`.<sup>21</sup>

There are various further refinements you could make, such as when the program/app starts up, you could read the default value of the Slider and update the display (the Static Text box)<sup>22</sup>. You might also add a Push Button to close the app.

---

As another test of your programming and use of *conditionals* (e.g. `ifs`) ... if the value of `bananaindex` goes above 5, set the text box background to blue. And if the value of `bananaindex` is below (or equal to) 5, set the color to red.<sup>23</sup>

<sup>21</sup> Remember that you cannot display a number directly where a string is required – use `num2str`.

<sup>22</sup> Code going in the *function*, `OpeningFcn`.

<sup>23</sup> The Static Text box property is called `ForegroundColor`. To set, e.g. add the code:

```
set(h_text2,'ForegroundColor',[1  
0 0]);
```



## 9.2 *MATLAB apps*



10

*Numerical modelling meets GUI (prettier games!)*

280 str = 'do you like bananas?'

## 10.1 GUI Pokémon 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 (Pokémon) 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 10.1, 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:

*Part 0* (Some graphics tricks.)

*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**) is provided on the course webpage if needed for guidance (e.g. what code goes where?). Example images are also provided (download via the course webpage) but 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`.

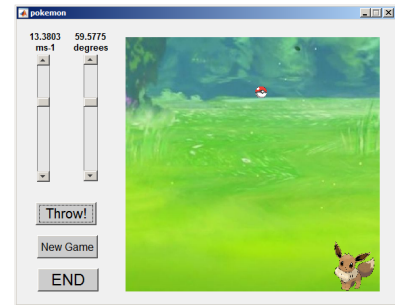


Figure 10.1: Screen-shot of the Pokémon game App.

**Part 0** – A few of the graphics procedures you will need to grasp and implement.<sup>1</sup>

Firstly, at the command line, open a Figure window (» `figure`). Download any (legal/moral) image you care from the internet<sup>2</sup>. You can load this image into the **MATLAB** workspace with `imread`, and display it in the Figure window with `imshow`. Try it. This fills up the screen, which is OK for a background image, but not for much else.

Open up a new Figure window.

You can define a set of axes anywhere in the window you like via the axes function:

```
axes('pos',[x,y,dx,dy])
```

where  $(x,y)$  are the co-ordinates in the window, which by default are from 0 – 1 in both  $x$  and  $y$  directions.  $dx$  and  $dy$  are the width and height, respectively, of the axes (in the same window coordinate system).

For instance, to create a set of axes starting at the origin, but only 25% of the full width and height of the window, type:

```
» axes('pos',[0.0,0.0,0.25,0.25]);
```

If you now display the image:

```
» imshow(A);
```

<sup>3</sup> you should see a smaller version of the image, positioned at the origin. If you assign the handle returned by `imshow` to a variable, by typing:

```
» h = imshow(A);
```

you can now delete/remove the image again:

```
» delete(h);
```

OK – now dig up the *script* for your ball-throwing animation – the one where the `scatter` plotting ball location symbol was deleted after a pause (giving the impression of movement/animation)<sup>4</sup>. Copy and rename the **m-file**. Near the start of the *script* (before the loop starts), load in the Pokéball graphic<sup>5,6</sup>. Then, instead of using the `scatter` function to plot a single point (circle), you are going to:

1. Define an axes object, either centered (harder) on the position of the ball that `scatter` plotted, or taking as its origin (easier), the position of the ball. The width and height of the axis ... you can play about with, but it should be a relatively small proportion of the total size of the main axes.

Note ... that `axes` uses relative coordinates (i.e. 0 – 1 in both dimensions) and not your actual ball position (in units of  $m$ ). So

<sup>1</sup> Example codes provided

<sup>2</sup> With the raster graphics format being one of: .jpg, .png, .tif.

```
imread
'A = imread(filename) reads
the image from the file specified by
filename ...'
and in this definition, assigns the
result of imread to a variable A.
```

```
imshow
imshow(A) will display an image
held in the variable A (read in by
imshow).
Assign the result of imshow to a
handle if you wish to do anything
with it later, i.e.
h = imshow(A);
```

<sup>3</sup> Or whatever you called the variable with the image in.

<sup>4</sup> From Part IV of Section 8.1

<sup>5</sup> Under **got data?** on the website.

<sup>6</sup> (or pick your own graphic)

you'll need to determine the horizontal and vertical position of the ball – as a fraction – of the total size of your domain.<sup>7</sup>

It is important not to be confused between the different sets of axes here – you defined the primary one, outside of the loop, and this defines the domain in which the trajectory is simulated, perhaps something like:

```
axes('Position',[0 0 1 1],'Visible','off');
```

For scatter-plotting, you then specified what  $(x,y)$  limits the axes represented, e.g.:

```
axis([0 x_max 0 y_max]);
```

(here, using parameters containing the maximum  $x$  and  $y$  limits in placed of fixed / hard-coded values).

Now, instead of scatter-plotting the position of the ball – within the loop, you are defining a small axes region to contain the image. (And whose location and width/height are given in relative  $(0 - 1)$  scale) units, rather than  $0 - x_{max}$  and  $0 - y_{max}$ .)

2. Plot the ball image (`imshow`). Assign the graphics *handle* returned by the function to a variable.
3. As per previously, after a delay, you can delete the graphic object,

Omitting `delete(h)`, the output of your ball/trajectory model should look like Figure 10.2.

Ignoring the fact that image deleting is disabled, the images (*sprites*) Figure 10.2 have a black background around them. Yuk. If you picked an image with a white background, it would look better, unless you had a dark background.<sup>8</sup>

Some (raster) graphics formats enable a transparency to be defined – basically a color that ... is transparent. Common formats with such a capability include `.gif` and `.png`. As `.png` is a valid format for `imread` – try and find a `.png` image on the internet with a transparency.<sup>9</sup> You could also use the Pokéball image provided (which has a transparent background).

To enact a transparent background in **MATLAB**, you first have to obtain additional handles when you read in the image:

```
[img_ball, h_map_ball, h_alpha_ball] = imread('Pokeball.png');
```

where `img_ball` is the variable containing the ball image, as before, and `h_alpha_ball` is a handle to ... lets not worry about what it is to. Just that you need it.

When you plot the ball, now add an additional command:

```
h = imshow(img_ball);
set(h, 'AlphaData', h_alpha_ball);
```

<sup>7</sup> e.g. you might have considered a maximum horizontal distance of  $10m$  and a maximum vertical distance of  $7.5m$  and specified:

```
axis([0 10 0 7.5]);
```

In which case, for the position of the ball in relative/normalized units – divide the  $x$  position by 10 and the  $y$  position by 7.5.

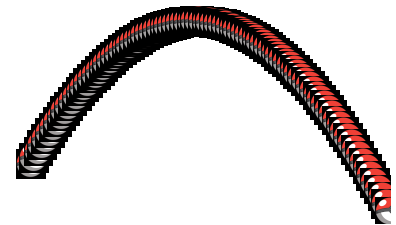


Figure 10.2: Trajectory model, with a Pokéball image replacing the `scatter` point. Here show without deleting the image once displayed.

<sup>8</sup> You could pick an image which is square. But what balls have you seen that are cubes? Seriously. Do you get out at all an play any sports? Or even watch TV?!??

<sup>9</sup> In Google search / images, a transparent background is illustrated as a grey-white checkerboard.

which sets this thing I am not telling you about.<sup>10</sup>

For instance – the same model as before (with the Pokéball replacing scatter but with the use of `delete`), but with only 1s simulated, plus a background image displayed (before the loops starts), and ... with a transparency implemented (removing the square black background), looks like Figure 10.3.

Now ... we are ready ...

---

### Part I – the basic GUI.

To achieve a GUI along the lines of Figure 10.1 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.<sup>11</sup>
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.

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 opened up<sup>12</sup>. **MATLAB** then opens the Editor and the GUI code template.

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 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.<sup>13</sup> 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.

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

<sup>10</sup> Wikipedia (please donate!), says: 'In computer graphics, alpha compositing is the process of combining an image with a background to create the appearance of partial or full transparency.' Without alpha channel information, everything is assumed 100% opaque (including the background).



Figure 10.3: Trajectory model (exactly the same trajectory as per the Figure 10.2), frozen mid-flight at  $t = 1$ s with the Pokéball passing over UC-Riverside.

<sup>11</sup> This we'll only worry about making use of this in Part IV.

<sup>12</sup> **File** – **Save As...**

<sup>13</sup> 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.

window. Click on the Axes icon and drag out the position and size of the object you want.<sup>14</sup> By default, it is assigned a name (its Tag property) of `axes1`. You are not going to have so desperately many axes objects that it is necessarily worth re-naming it, but you can if you wish. 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 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<sup>15</sup>. 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<sup>16</sup>, and nothing else.

2. Next: *Push Button #1*. Create (position and size, where- and how-ever you think best). Simplest is to leave the default name ('`pushbutton1`'). 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. This will be 'New game' (or equivalent).
4. Similarly, create 3rd *Push Button* ('`pushbutton3`'). This will be the end/close game (and window) button.
5. Now we need a *Slider*<sup>17</sup> bar. These are bars 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`').

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 maxi-

<sup>14</sup> 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.

<sup>15</sup> 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.

<sup>16</sup> We could remove these black lines, but they'll get covered up later.

<sup>17</sup> Not anything to do with baseball.



mum property value name are Min and Max – edit these to span a plausible initial speed range<sup>18</sup>. Also set a default initial value (parameter name 'Value')<sup>19</sup>.

6. Create a second Slider ('slider2') for setting the initial angle of the ball (*theta*).<sup>20</sup>

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 10.1, 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 Pokémon 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:

<sup>18</sup> I used 0 to  $20\text{ms}^{-1}$ .

<sup>19</sup> I assumed  $0\text{ms}^{-1}$ .

<sup>20</sup> Here I assumed a range of 0 to  $90^\circ$ , with a default of  $0^\circ$ .

```
% -- 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 of these two functions (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 following (automatically generated) lines:

```
% Choose default command line output for Pokémon
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
% UIWAIT makes Pokémon wait for user response (see
UIRESUME)
% uiwait(handles.figure1);
```

First, we'll read in a background image (e.g. background.jpg – available for download from the course webpage, or pick your own) 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' to display it.<sup>21</sup> We can get the *handle* (ID) of the Axes object via:

```
h_axes = findobj('Tag','axes1');
```

and then tell **MATLAB** that this is currently the object to put things in by:

```
axes(h_axes);
```

(which sets the current/active axes object to the one with the handle `h_axes`). We then use this handle in the call to `imshow`:

<sup>21</sup> Actually, it may work without worrying about this, but we'll need to be able to specify where to position other images later anyway.

```
288 str = 'do you like bananas?'
```

```
h_background = imshow(img_background, 'Parent', h_axes);
```

Add this code and try it (i.e. run your App). (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 `function` `Pokemon_OpeningFcn`), 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');
% find handle of axes object
h_axes = findobj('Tag', 'axes1');
% make axes object the current axes
axes(h_axes);
% set axis size to scale image to
axis([0 x_max 0 y_max]);
% get ready to draw ...
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 are explicitly prescribed (and the image scaled to completely fill the axes object).

When you run all this, you should get Figure 10.4.

---

Next, we want a Pokémon to throw the ball at! The load-in code for the image is identical to before:

```
img_eevee = imread('Eevee.png');
```

and the code goes immediately after you have drawn the background image. (The image itself ('Eevee.png') can be downloaded from the course webpage ... or download your own ...) 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);
```

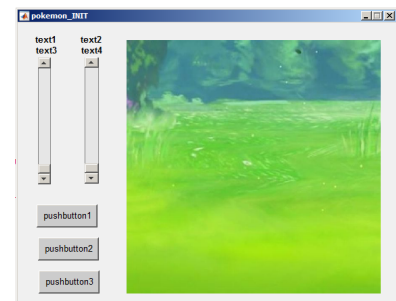


Figure 10.4: Template App with background image.

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 Pokémon 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 Pokémon 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 10.5.

You can see firstly that the Pokémon 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 ... in the Axes object, it seems that the  $y$ -axis origin starts at the top and is positive downwards (which is why the Pokémon appears in the top left, rather than bottom left, corner).

To cut a long story short, we can generalize the position and size of the Pokémon that is displayed (and use this at the end when we refine the App), via the following code fragment<sup>22</sup>:

```
% define Pokemon size
dx_Pokemon = 0.2*x_max;
dy_Pokemon = 0.2*y_max;
% define initial Pokémon position
x_Pokemon = x_max-dx_Pokemon;
y_Pokemon = y_max-dy_Pokemon;
% read in Pokemon image
img_eevee = imread('Eevee.png');
% draw Pokémon
h_eevee = imshow(img_eevee, 'Parent', h_axes, 'Xdata', [x_Pokemon...
x_Pokemon+dx_Pokemon], 'Ydata', [y_Pokemon-dy_Pokemon
y_Pokemon]);
```

Now giving you a small Pokémon – in fact, 20% of the Axes size as specified in the definition of the Pokémon size parameters, `dx_Pokemon` and `dy_Pokemon`. If you run this, you should get Figure 10.6. (Note that because  $y$  is measured downwards from the top in the GUI Axes object, for `'Ydata'`, we write the  $y$  min and amx values the other way around: `[y_Pokemon-dy_Pokemon y_Pokemon]`.)

One final thing now is the background to the Pokémon image. The original format (png) is 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');
```

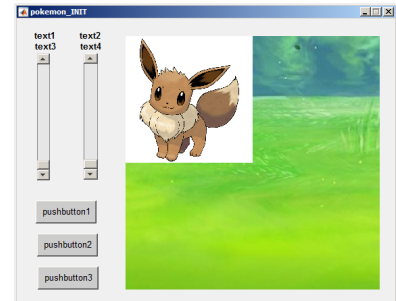


Figure 10.5: Template App with background image plus Pokémon.

<sup>22</sup> 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.

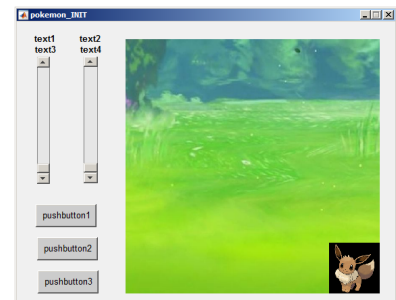


Figure 10.6: Template App with background image plus small Pokémon at bottom right.

290 str = 'do you like bananas?'

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 – Figure 10.7.

---

### Part III – incorporating the ballistics model.

Here – almost all the code for the animated ballistics model 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).<sup>23</sup>

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 been done within the initialization function.

```
% model constants
g = 9.81;
% model parameters
theta0 = 80.0;
s0 = 5.0;
h0 = 2.0;
% 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;
% 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
    dv = -dt*g;
    v = v + dv;
end
```

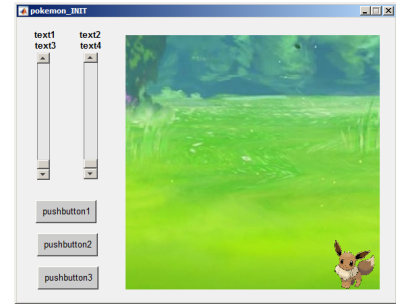


Figure 10.7: Template App with background image plus small Pokémon at bottom right, now with its transparency applied.

<sup>23</sup> Remember – double-click on the `pushbutton1` object in the design editor and then find and edit the value of the **String** property.

Either copy-paste this or add your own version (and comment out the figure creation line. Note that this is a simplified code using `scatter` rather than plotting a sprint – to keep things simple and to aid in debugging.

When you run 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 the start of both the following functions:

- `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 such as comment lines).

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 as per Section 0 of this chapter.

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

292 `str = 'do you like bananas?'`

`x_axes` as `global`<sup>24</sup>, 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't worry about it, after much trial-and-error, it works. Now you should have something like Figure 10.8 when you run it.

To finish this section off, we'll improve how the trajectory of 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<sup>25</sup> use the `get` function to get the

<sup>24</sup> Directly underneath the other two `global` definition lines AND in a similar position in the initialization function: `function pushbutton1_Callback .`

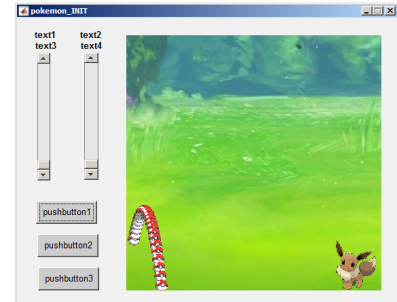


Figure 10.8: App with ball trajectory trail.

<sup>25</sup> On the next line.



'value' (one of the properties of the object):

```
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`<sup>26</sup>.

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 `s0` 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.

<sup>26</sup> At least, it was in my GUI design – check the name of yours.

294 str = 'do you like bananas?'

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, e.g.:

```
%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 its 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<sup>27</sup>.

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 (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, add the appropriate lines to make `s0` and `theta0` global too.

<sup>27</sup> 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.

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.<sup>28</sup>

---

### Part V – pokéball/Pokémon collision detection.

How are you going to tell if the throw is successful or not? Remember earlier – you detected if the height of the ball fell below ground level and used this to exit the loop (because no more calculations were necessary), as so:

```
if (y < 0.0)
    break;
end
```

You are going to do something similar, but:

1. Firstly, test both  $x$  and  $y$  positions of the ball (rather than just the value of  $y$ ).
2. Finish the game upon a successful 'hit' (collision detection).

For the first part – you need to determine whether the ball is within the limits of the Pokémon (which would be a reasonable criteria for a 'hit'). There are four parts to the criteria, which all need to be *true*:

1. The ball center is to the right of the left edge of the Pokémon.
2. The ball center is to the left of the right edge of the Pokémon.
3. The ball center is above the bottom edge of the Pokémon.
4. The ball center is below the top edge of the Pokémon.

(We could make it more complicated and more realistic if we also considered the size of the ball, in which case (1) for instance, would become: 'The right edge of the ball is to the right of the left edge of the Pokémon'.)

In code, if the edges of the Pokémon are:

```
xmin, xmax, ymin, ymax
```

we are looking for the situation:

```
x>xmin && x<xmax && y>ymin && y<ymax
```

where  $(x, y)$  is the location of the ball.<sup>29</sup>

For the edges of the Pokémon – refer back to the code in `function Pokemon_OpeningFcn` where you defined the position of the Pokémon image. The only thing is to remember the up-side-down  $y$ -axis, so you are actually looking for  $y$  to be greater than the top edge, and less than the bottom edge ...

If this condition is met, the game is over. You might then:

<sup>28</sup> 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.

<sup>29</sup> Note that the code you need is not quite this simple – your ball  $(x, y)$  location is in units of  $m$ , with  $y$  positive upwards, whereas the Pokémon image location and size is defined in normalized Axes units, and with  $y$  downwards.

296 str = 'do you like bananas?'

- Remove the Pokémon image and replace with a message.
- Grey out and disable the 'Throw' button.

See if you work out these 2 pieces of code.

---

### *Part VI* – final game refinements.

Various refinements that come to mind and that you might try and implement:

- Upon clicking 'New Game', you might place the Pokémon in a different place. Perhaps larger or smaller than originally. Both these settings could be made random.
- In a new game, you might display a different Pokémon. Which Pokémon gets displayed, could also be random.
- Keep score (how many 'catches') as well as how many tries total. This would require two new Static Text box objects in the GUI.
- You could also keep a high score ... saving this value when you close the App, and loading it when you start it up.  
(e.g. simply saving and loading an integer from a .mat file.  
Harder, is to add the ability to enter (and remember) the initials of the person with the high score ...

**11**

*Example codes*

298 `str = 'do you like bananas?'`

11.1 *Chapter 1 codes*

## 11.2 Chapter 2 codes

## Section 2.4

Code for loading, storing, and plotting, monthly global temperatures

```

% *****
% Program to load in 12 monthly temperature data-sets,
% store the data in a 3D array,
% and plot each monthly global temperature distribution
% *****
% close all currently open figure windows
close all;
% START OF MONTHLY LOOP
for month=1:12
    % create filename
    filename = ['temp' num2str(month) '.tsv'];
    % load data and assign to a new array 'slice'
    temp(:,:,month) = load(filename);
    % create new figure window and plot data slice
    figure;
    pcolor(temp(:,:,month));
end
% Close the file
close(vidObj);
% END OF MONTHLY LOOP
% *****

```

Code for creating an animation

```

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

```

```
300 str = 'do you like bananas?'
```

## Section 2.4

*Code for loading and plotting the continental outline*

```
%%% code to plot the continental outline %%%
% load data files
co_k = load('continental_outline_k.dat','-ascii');
co_start = load('continental_outline_start.dat','-ascii');
co_end = load('continental_outline_end.dat','-ascii');
co_lat = load('continental_outline_lat.dat','-ascii');
co_lon = load('continental_outline_lon.dat','-ascii');
% determine number of line segments
n_lines = length(co_k);
% create new figure window and set axes
figure;
axis([-180 +180 -090 +090]);
% 'hold on' ...
hold on;
% LOOP ... and draw thin line segments
for k = 1:n_lines
    plot(co_lon(co_start(k):co_end(k)), ...
         co_lat(co_start(k):co_end(k)), 'k-', 'LineWidth', 0.25);
end
% adjust the plot aspect ratio
set(gca, 'PlotBoxAspectRatio', [1.0 0.5 1.0]);
% draw a nice boundary to the map
h = plot([-180 +180], [-90 -90], 'k-');
set(h, 'LineWidth', 1.0);
h = plot([-180 +180], [+90 +90], 'k-');
set(h, 'LineWidth', 1.0);
h = plot([-180 -180], [-90 +90], 'k-');
set(h, 'LineWidth', 1.0);
h = plot([+180 +180], [-90 +90], 'k-');
set(h, 'LineWidth', 1.0);
% 'hold off' (not strictly necessary)
hold off;
% label plot
xlabel('longitude', 'fontsize', 15);
ylabel('latitude', 'fontsize', 15);
title('Continental outline', 'fontsize', 18);
% export graphics (as postscript)
print -dpsc2 ch3p2p6.ps;
```



### 11.3 Chapter 4 codes

#### Section 4.2

Code for the `maxxx` function

```
function [s_out] = maxxx(v_in)
% maxxx
%
% Takes a (single) vector as input,
% returns the maximum value.
% Determine number of elements in vector
nmax = length(v_in); % Seed temporary (running maximum)
variable
temp_max = v_in(1);
% Loop through all
% but the first element in the vector
for n = 2:nmax,
    if (v_in(n) > temp_max),
        temp_max = v_in(n);
    end
end
% Set function (return) value
s_out = temp_max;
end
```

```
302 str = 'do you like bananas?'
```

## 11.4 Chapter 10 codes

### Section 10.0

Code for the sprite (image) trajectory animation

```
% load images
img_background=imread('DSC01182.JPG');
[img_ball, h_map_ball, h_alpha_ball] = imread('Pokeball.png');
% set parameter values controlling trajectory
g = 9.81;
theta = 60.0;
s0 = 10.0;
h0 = 1.0;
% set parameter values controlling time-stepping
dt = 0.02;
t_max = 2.0;
% derive initial velocity components
u = s0*cos(pi*theta/180.0);
v = s0*sin(pi*theta/180.0);
% set initial position
p(1) = 0.0;
p(2) = h0;
% create window and scale axes
figure;
x_max = 10;
y_max = 7.5;
axes('Position',[0 0 1 1],'Visible','off');
h = imshow(img_background);
hold on;
% time-stepping update of position and graphics
for t=dt:dt:t_max
    dp = dt*u;
    p(1) = p(1) + dp;
    dp = dt*v;
    p(2) = p(2) + dp;
    axes('pos',[p(1)/x_max,p(2)/y_max,0.1,0.1])
    h = imshow(img_ball);
    set(h, 'AlphaData', h_alpha_ball);
    pause(0.001);
    delete(h);
    if (p(2) < 0.0)
        break
    end
    dv = -dt*g;
    v = v + dv;
end
```

## *Bibliography*

Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Pocket Books, 1979. ISBN 0-671-46149-4.



# Index

- .mat environment, 46
- ; environment, 27
- = environment, 25, 26, 28
  
- addition environment, 27
- addpath environment, 44
- and environment, 29
- axis environment, 35, 36, 51
  
- break environment, 90
  
- cell array environment, 102, 104
- cell2mat environment, 102, 104
- clabel environment, 111, 112
- clear all environment, 30
- clear environment, 30
- close all environment, 30
- close environment, 30
- colon operator environment, 31, 33, 39
- color operator environment, 32
- colorbar environment, 112, 134
- colormap environment, 109, 138
- Command Window, 22
- comment symbol environment, 50
- comments environment, 102
- contour environment, 108, 111
- contourf environment, 108
  
- D'uh
  - D'uh environment, 234
- disp environment, 62, 85
- division environment, 27
- duh environment, 234
  
- else environment, 71
- elseif environment, 71
- end environment, 33
  
- environments
  - .mat, 46
  - i, 27
  - =, 25, 26, 28
  - addition, 27
  - addpath, 44
  - and, 29
  - axis, 35, 36, 51
  - break, 90
  - cell array, 102, 104
  - cell2mat, 102, 104
  - clabel, 111, 112
  - clear, 30
  - clear all, 30
  - close, 30
  - close all, 30
  - colon operator, 31, 33, 39
  - color operator, 32
  - colorbar, 112, 134
  - colormap, 109, 138
  - comment symbol, 50
  - comments, 102
  - contour, 108, 111
  - contourf, 108
  - disp, 62, 85
  - division, 27
  - duh, 234
  - else, 71
  - elseif, 71
  - end, 33
  - equality, 28
  - errorbar, 53, 54
  - exist, 91, 94
  - exit, 30, 216
  - exponentiation, 27
  - fclose, 100
  - figure, 34
  - find, 118–120, 123
  - fliplr, 32, 40
  
- flipud, 32
- flipup, 40
- fopen, 100–102
- for, 78
- fprintf, 44
- FUNCTION, 17
- gca, 133
- geoshow, 116
- getframe, 86
- ginput, 189
- greater than, 28
- greater than or equal to, 28
- help, 17, 23
- hold, 53, 54
- hold on, 51
- icecream, 164
- if ... end, 71
- image, 56, 108
- imagesc, 108, 109
- imread, 56, 281
- imshow, 281
- inequality, 28
- input, 71, 72
- interp1, 128
- isempty, 196
- ismember, 197
- isnan, 122, 123, 125
- isnanana, 122
- legend, 54
- length, 32, 97, 151
- less than, 28
- less than or equal to, 28
- line, 132, 206, 244
- load, 43–45, 47
- m-files, 35
- max, 118
- mean, 249
- meshgrid, 113
- min, 118

- `mod`, 148
- `movie2avi`, 87, 88
- `multiplication`, 27
- `NaN`, 120
- `ncread`, 105
- `not`, 29
- `num2str`, 85
- `numel`, 182
- `or`, 29
- `patch`, 135
- `pause`, 237
- `pcolor`, 55, 56, 95
- `pi`, 29
- `plot`, 34, 48, 53
- `print`, 36
- `quiver`, 135
- `rand`, 147
- `reshape`, 126
- `rmmissing`, 124
- `rocker`, 148
- `rotate`, 40
- `round`, 191
- `save`, 45, 46, 172
- `scatter`, 34, 55
- `set`, 133, 134, 273
- `sin`, 36
- `size`, 32, 39, 163
- `sort`, 48
- `sortrows`, 48
- `strcmp`, 73
- `subplot`, 36
- `subtraction`, 27
- `sum`, 40
- `switch ... case ... end`, 76
- `text`, 137
- `textscan`, 100–103
- `title`, 35, 36
- `transpose`, 40, 109
- `transpose operator`, 32, 33
- `VideoWriter`, 87
- `VideoWriter`, 87
- `while`, 78
- `xlabel`, 35
- `xlsread`, 104
- `ylabel`, 35
- `zeros`, 148, 163, 210
- `equality environment`, 28
- `errorbar environment`, 53, 54
- `exist environment`, 91, 94
- `exit environment`, 30, 216
- `exponentiation environment`, 27
- `fclose environment`, 100
- `figure environment`, 34
- `find environment`, 118–120, 123
- `fliplr environment`, 32, 40
- `flipud environment`, 32
- `flipup environment`, 40
- `fopen environment`, 100–102
- `for environment`, 78
- `fprintf environment`, 44
- `FUNCTION environment`, 17
- `gca environment`, 133
- `geoshow environment`, 116
- `getframe environment`, 86
- `ginput environment`, 189
- `greater than environment`, 28
- `greater than or equal to environment`, 28
- `help environment`, 17, 23
- `hold environment`, 53, 54
- `hold on environment`, 51
- `icecream environment`, 164
- `if ... end environment`, 71
- `image environment`, 56, 108
- `imagesc environment`, 108, 109
- `imread environment`, 56, 281
- `imshow environment`, 281
- `inequality environment`, 28
- `input environment`, 71, 72
- `interp1 environment`, 128
- `isempty environment`, 196
- `ismember environment`, 197
- `isnan environment`, 122, 123, 125
- `isnana environment`, 122
- `legend environment`, 54
- `length environment`, 32, 97, 151
- `less than environment`, 28
- `less than or equal to environment`, 28
- `license`, 2
- `line environment`, 132, 206, 244
- `load environment`, 43–45, 47
- `m-files environment`, 35
- `max environment`, 118
- `mean environment`, 249
- `meshgrid environment`, 113
- `min environment`, 118
- `mod environment`, 148
- `movie2avi environment`, 87, 88
- `multiplication environment`, 27
- `NaN environment`, 120
- `ncread environment`, 105
- `not environment`, 29
- `num2str environment`, 85
- `numel environment`, 182
- `or environment`, 29
- `patch environment`, 135
- `pause environment`, 237
- `pcolor environment`, 55, 56, 95
- `pi environment`, 29
- `plot environment`, 34, 48, 53
- `print environment`, 36
- `quiver environment`, 135
- `rand environment`, 147
- `reshape environment`, 126
- `rmmissing environment`, 124
- `rocker environment`, 148
- `rotate environment`, 40
- `round environment`, 191
- `save environment`, 45, 46, 172
- `scatter environment`, 34, 55
- `set environment`, 133, 134, 273
- `sin environment`, 36
- `size environment`, 32, 39, 163
- `sort environment`, 48
- `sortrows environment`, 48
- `strcmp environment`, 73
- `subplot environment`, 36
- `subtraction environment`, 27
- `sum environment`, 40
- `switch ... case ... end environment`, 76
- `text environment`, 137
- `textscan environment`, 100–103
- `The command line`, 22
- `title environment`, 35, 36
- `transpose environment`, 40, 109
- `transpose operator environment`, 32, 33

typefaces  
  sizes, 57

variable, 24

VideoWriter environment, 87

VideoWriter, environment, 87

while environment, 78

xlabel environment, 35

xlsread environment, 104

ylabel environment, 35

zeros environment, 148, 163, 210