ANDY RIDGWELL

GEO111 – NUMERICAL SKILLS IN GEOSCIENCE

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

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

First printing, June 2016

Contents

1 *MATLAB basics* 17

2 *Plotting and visualizing data* 33

3 *MATLAB scripting and programming* 49

4 *Introduction to numerical modelling* 73

5 *1- and 2-D numerical modelling* 93

Bibliography 111

Index 113

List of Figures

1 weeks 1-5 13
2 weeks 6-10 14

1.1 Direct sea-level data (210 kyr to present) from corals (solid symbols) and speleothems (gray symbols). July summer solar insolation at 65 degrees north latitude is depicted in the top panel. This is thought to control the glacial-interglacial variations in the size of the Norther Hemisphere ice sheets. The vertical gray bars indicate times of high insolation and the correspondence with minima in ice volume (equivalent to maxima in sea-level). (from; Holland, H. D and K. K. Turekian, eds, Treatise on Geochemistry, Elsevier, 2004) 28

2.1 Default output of `plot`. 33
2.2 Past sealevel variability as reconstructed from oxygen isotopes. 34
2.3 proxy reconstructed past variability in atmospheric CO2. 35
2.4 Proxy reconstructed past variability in atmospheric CO2 (sorted data). 36
2.5 Proxy reconstructed past variability in atmospheric CO2 (sorted data). 36
2.6 Proxy reconstructed past variability in atmospheric CO2 (sorted data). 37
2.7 Arrangement of subplots. 37
2.8 Proxy reconstructed past variability in atmospheric CO2 (scatter plot). 38
2.9 Proxy reconstructed past variability in atmospheric CO2 (scatter plot). 38
2.10 Proxy reconstructed past variability in atmospheric CO2 (scatter plot). 39
2.11 Proxy reconstructed past variability in atmospheric CO2 (scatter plot). 40
2.12 Very basic imaging (`image`) of an array (2D) of data – here, global bathymetry. 44
2.13 Slightly improved very basic imaging (`imagesc`) of bathymetry data. 44
2.14 **Example contour plot.** Result of `contour(data,20)`, where the data file was `temp7.tsv`. 44
2.15 **Example contour plot.** Result of `contourf(lon,lat,temp7,30)`, where the data file was `temp7.tsv`, with some embellishments. 46

3.1 Output from the `plot_some_dull_stuff` **m-file**. 51
3.2 Continental outline (of sorts). 57
3.3 Another continental outline (of sorts). 57
3.4 Another go at the continental outline! 58
3.5 Now continents on top of temperature fields. 60

3.6 Ocean topography (blues through red) in the 'GENIE' Earth system model. Land is shown in brown. 65
3.7 The 'GENIE' mode land grid, with land points assigned a sequential integer (working across and dow the grid – from West to East, and then North to South). 67
3.8 The 'GENIE' mode land grid, with land points assigned a unique identifier ... almost ... (!) 71
3.9 The 'GENIE' mode land grid, with land points assigned a unique identifier (color). 71
3.10 The 'GENIE' mode land grid, with land points (almost) assigned a unique identifier (color). 72
3.11 The 'GENIE' mode land grid, with land points assigned a unique identifier (color). 72

4.1 Lake volumes and river flow rates in the Great Lakes system. 77
4.2 Simulated evolution of metal concentration in the Great Lakes system with time ... with labels that are far too small to make out :o) 83
4.3 Simulated evolution of metal concentration in the Great Lakes system with time ... with labels that are far too small to make out ... and an integration time-step that is too long. 84
4.4 Simple EBM projection of the evolution of Earth surface temperature with time. 85
4.5 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). 87
4.6 Evolution of global surface temperature and the two populations of daisies with time ... but now assuming that the growth of each depends only on the global mean surface temperature. Symbols as per Figure 4.5. 88
4.7 Evolution of global surface temperature and the two populations of daisies with time. Symbols as per Figure 4.5. 88
4.8 100 yr spin-up of the basic EBM. 89
4.9 100 yr spin-up of the basic EBM, but with a poor choice of time-step ... 89
4.10 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. 90
4.11 Transient EBM response to (fake) changes in atmospheric CO_2. 91

5.1 Basic 1-D EBM with no latitudinal heat transport. 95

5.2 Basic 1-D EBM with no latitudinal heat transport (red filled circles). Overlain is the zonal mean observational data for January (blue circles). 96
5.3 As per Figure 5.2 but for July. 96
5.4 1D EBM with an initial guess as to the value of . 98
5.5 1D EBM with a larger value of . 98
5.6 Idealized schematic of the soil-CH_4 system. 99
5.7 Slightly less idealized schematic of the soil-CH_4 system. 99
5.8 Even less idealized and almost realistic, schematic of the soil-CH_4 system. 99
5.9 Soil profile of CH_4 after 10.0s of simulation. 106
5.10 Soil profile of CH_4 after 100.0s of simulation. 106
5.11 Soil profile of CH_4 after 100.0s of simulation with an extremely marginal choice of time-step length. 106
5.12 Soil profile of CH_4 after 100.0s of simulation, with CH_4 uptake at the base of the profile with a rate constant of 1.0 per s. 107
5.13 Equilibrium soil profile of CH_4, with CH_4 uptake throughout the soil column with a rate constant of 0.1 per s. 108
5.14 Example equilibrium soil profile of CH_4 with production at depth. 108

List of Tables

4.1 Pollution input input rates to each of the 5 lakes. 77

Introduction

GEO111 will provide an introduction to computer programming and numerical modelling for Earth and Environmental Science problems. It will provide a chance to learn a computer programming language and all the elements that constitute it, including concepts in number bases and types, logical constructs, debugging, etc. The course will develop programming skills step-wise, applying them at each point to practical questions and outcomes, such as data processing and visualization. How complex environmental processes can be encapsulated and approximated, and numerical models thereby constructed, will be illustrated. Guided opportunities will be provided to build a 'DIY' climate model and in doing so, further develop programming and modelling skills at the same time as reinforcing basic concepts in climate dynamics through practice in addition to theory.

The cumulating objectives of the course are to:

1. develop an understanding of how computers and the internet work and hence foster a critical understanding of modern technology,
2. provide hands-on training in how computer programs are written and numerical models constructed, and
3. develop both general (transferable) as well as specific numerical and analytical skills applicable to the Earth and Environmental Sciences.

The associated learning goals are firstly; to provide, through hands-on practical exploration, factual knowledge and an understanding of:

- Number bases, how computers work plus computer programs and their basic building blocks. (Learning Outcome 2).
- Numerical models and the representation of time. Construction and application of a variety of models spanning box models of biogeochemical cycles and population dynamics, through 1D reaction-diffusion models of surface Earth processes, to 3D gridded global models. (Learning Outcomes 1 and 2).
- The Greenhouse effect and basic climate feedbacks. (Learning Outcome 1).
- Awareness of different operating systems such as linux; of compiled languages such as FORTRAN, plus how webpages and the internet work. (Learning Outcome 2).
- The use of numerical models in addressing scientific questions and testing hypotheses as well as the limitations of numerical models. (Learning Outcomes 2 and 4).

and provide transferable skills in

- Written communication and presentation. (Learning Outcome 3).

- Problem solving and quantitative analysis together with logic and fault-finding. (Learning Outcomes 4 and 5).
- Computer programming. (Learning Outcomes 2 and 4).
- Effective internet use and website construction. (Learning Outcomes 2 and 4).

0.1 Course logistics

0.1.1 Format

The weekly format of the Class is: one 1-hour lecture, one 3-hour computer practical session, plus a 2-hour interactive lecture/discussion session of worked problems and examples. The computer practical class is the central element, and will consist of structured exercises leading step-by-step through the components of computer programming and numerical model construction, debugging, and testing, plus applications to common geosciences problems. The lecture starting each week will outline the basics and introduce the key concepts of the week. The purpose of the 2-hour lecture/discussion session ending the week is to ensure all the concepts are understood and misconceptions resolved and will be a mix of presentation and worked-through examples, plus questions and discussion.

0.1.2 Timetable

0.1.3 Assessment

The course will be assessed as follows:

- Midterm paper – 40%
- Finals paper – 60%

(Part of the intention of the shorter Midterm assessment, being to provide critical feedback and guidance for the main Finals paper.)

The mid-term paper will consist of a computer model written in MATLAB to solve a specific problem, with the code to be handed in. The code will be accompanied by a 'user manual' for running the model code and conducting experiments. As part of the assessment, the code will be run to help judge the overall success of the program, with additional marking of the code itself (including structure and commenting). This will together constitute 40% of the total assessment of the course.

The Finals paper will be in the form of a science paper describing the model, its evaluation, application to a specified science question, plus discussion of model caveats and suggestions for future improvements. The scope of the model exercise will be somewhat restricted

	Monday am (1)	Monday am (2)	Friday pm
WEEK	***Lecture A*** ***08:10-09:00*** ***GEOL 1444***	***Computing lab*** ***09:10-12:00*** ***Watkins 2101***	***Lecture B*** ***14:10-16:00*** ***Watkins 2101***
01 / 28th March	**Introduction to the course** Format and content of the course. Office hours. Overview of course assessment. Introduction to MATLAB and the 'command line'. Numbers, vectors, and matricies.	**MATLAB basics** MATLAB basics, including variables and matrixes, data I/O. Data processing in MATLAB. Basic statistics.	**MATLAB basics** Continuation of the Lab Worked examples
02 / 4th April	**Computer hardware and software** Basic constituents and functioning of computers. Bases, logic and logic gates. Computer operating systems, programs, and software. Compiled and interpreted languages.	**Plotting data in MATLAB** Data visualization – types of 1-D plots. 2-D ploting and interpolation. Re-gridding. Data binning and histograms.	**Further visualization; Q&A.** Continuation of the Lab Worked examples
03 / 11th April	**Computer program fundamentals**	**Elements of MATLAB programming** Loops and conditionals. Subroutines and functions. Further data processing techniques.	**Further programming; Q&A** Algorithms and numerical techniques. Search and sort algorithms. Programming best practice and debugging. Functions and subroutines.
04 / 18th April	**Introduction to numerical models** Time-stepping and integration techniques. Numerical stability and accuracy. Model code structure.	**0-D ('box) modelling** Basic geochemical box modelling and reservoir dynamics. Isotopes and fractionation.	**Further numerical modelling; Q&A** Worked and literature examples of (bio)geochemical box modlling and how these are coded.
05 / 25th April	**1-D and reaction-transport models** Structure and ppplication of reaction-transport abd other 1-D models to environmental problems.	**1-D modelling** Example models: (1) gas diffusion and consumption in soils; (2) firm enclosure and signal filtering of records in ice cores.	***Discussion; work on MIDTERM***

Figure 1: Course schedule: weeks 1 through 5.

	Monday am (1)	Monday am (2)	Friday pm
WEEK	***Lecture A*** ***08:10-09:00*** ***GEOL 1444***	***Computing lab*** ***09:10-12:00*** ***Watkins 2101***	***Lecture B*** ***14:10-16:00*** ***Watkins 2101***
06 / 2nd May	**2-D global environmental models #1** Fundamental climate system processes and their representation in models.	**DIY climate model #1** Surface energy budget and greenhouse gases. Heat capacity. Atmsoperhic transport.	**Visualizing 2-D (and 3-D) models**
07 / 9th May	**2-D global environmental models #2** Ice dynamics and feedbacks.	**DIY climate model #2** Contorls on ice melt. Controls on ice flow. Feedbacks with climate.	**Evaluating numerical models** Evaluation of numerical models. Model-data assessment. Numerical models in the literature, the work of the IPCC, and model 'inter-comparisons'.
08 / 16h May	**Empirical and biological models** Derivation and application of empirical relationships. Geochemical empirical models. Biological empirical models. Modelling ocean biogeochemical cycles.	**Marine biogeochemical modelling** Matrix maths. Calculation of ocean circulation by MATLAB matrix maths. Creating a basic biogeochemical cycling model.	**Ecological modelling**
09 / 23rd May	**Complex models** Resolution and numerics. Parallelization.	**Global carbon cycling modelling** Creating a basic global carbon cycle model.	**Model dynamics** Perturbing and analysing models.
10 / 30th May	**The Internet** How the internet 'works'. Computer networks. TCP/IP and network communication.	**Basic html scripting** Creating a basic webpage.	***Discussion; work on FINALS***

Figure 2: Course schedule: weeks 6 through 10.

with a short menu of possible choices, but with considerable flexibility in terms of exactly what is done and explored with it (i.e. there is some slightly possibility of actually having fun!). This will constitute 60% of the total assessment of the course.

0.1.4 Office Hours

There are no specific Office Hours, but rather an open invitation to drop by[1] (excluding Thursdays) and/or email[2] questions. Part of the purpose of the lab session on Fridays is to provide an opportunity for further clarification of the course material and to go through worked examples.

[1] My office is in the Geology building, room 464 (basement floor).

[2] andy@sea02.org

0.1.5 Course text

There is no one (or even two between them) complete course texts that covers both basic computer programming and numerical modelling at a suitable level, and certainly not in the context of MATLAB. However, the recommended course textbook; *Matlab (Third Edition): A Practical Introduction to Programming and Problem Solving*[3], represents a good basis for the MATLAB part of the course. For additional reading, potential texts include:

[3] Stormy Attaway. *Matlab (Third Edition): A Practical Introduction to Programming and Problem Solving*. Butterworth-Heinemann, 2013

- *The Climate Modelling Primer (4th Edition)*, by Kendal McGuffie and Ann Henderson-Sellers. Wiley-Blackwell (2014). ISBN: 978-1-119-94336-5.
- *Introduction to MATLAB (3rd Edition)*, by Delores M. Etter. Prentice Hall (2014). ISBN: 978-0133770018.

Ultimately, the aim of the course (in future years) is to have a dedicated (free) text book in downloadable PDF format, of which the present document is the nucleus ...

1
MATLAB basics

Hello Newbies! This first lab's porpoise is to start to get you familiar with what MATLAB is all about and understand how to import and manipulate (array) data in this software environment. 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) on the screen. However, you will be much better off in the long run if you stick with the instructions and do everything using the command line that is asked of you (rather than doing stuff with the GUI instead). You will see why later in the course. 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.

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 least: *Command Window*. Depending on whether you have used MATLAB before and it has remembered your settings, windows may also include: *Command History*, *Workspace*, *Current Folder*. If instead you see; 'Tetris', 'Grand Theft Auto: San Andreas', and 'World Championship Pool', then you have the wrong software running and are going to find learning MATLAB rather hard. However, there is big $$$ to be made in on-line gaming tournaments these days. Would you really rather be a geologist and spend the rest of your days hitting rocks with a hammer? If so (you fool), read on ...

1.1.2 The command line

When MATLAB initially starts up, the *Command Window*[1] should display the following text:

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

```
Academic License
»
```

or in order versions of the software:

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

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

If you are unfamiliar with using command-line driven software ... Don't Panic! Nothing 'bad' can happen, regardless of what you do. Well, almost. It is possible to accidently clear MATLAB 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. But later on we will be placing the sequence of commands into a file (that is saved) and so ultimately, MATLAB should turn out to be mostly fool-proof.

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

1.1.3 MATLAB GUI

There are lots of fancy looking icons and pretty colors and you could spent all day staring at them and not getting any work done. Or learn good programming practice. Which is why we mostly will ignore the eye-candy and little (if any) guidance will be given as to the functionality of the GUI. Look at this as a lesson for the user (to read the Help, textbook, on-line documentation, or simple go Google for an answer[3]).

[3] i.e. Internet fishing

1.1.4 Help(!)

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

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

1.2 Basic concepts

1.2.1 Variables

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

[5] In the bad old days, this pointer was the actual address in memory.

a variable name based on 2 (or more) words, separated by an underscore (_). Valid variable names would include:

```
A
B
cat
derpyhooves
this_is_boring_stuff
BIG
big
```

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

Variables are entirely useless unless they have some information assigned to them. In fact, you can type in any of the variable names above (at the command line) and MATLAB will deny it knows what you are talking about[6].

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

So you need to assign something to it. Which brings us to quite 'what'. Variables can have the following *types*:

- **Integer** – An integer number is a counting number, i.e. `1, 2, 3, ...` and including zero and negative integers. MATLAB has different representations for integer numbers, depending on how large a number you need to represent (and how much memory it will need to allocated to storing it). This is something of a throwback to the days when computers only had $1/10000000^{th}$ of the memory of your iPhone and were slower than a lemon.

- **Real (floating point)** – A real number can have a non-integer component, e.g. 1.5 or $6.022140857 \times 10^{23}$. Real numbers also come in different precisions in MATLAB (also to do with memory allocation and speed), determining not just the number of decimal places that can be represented, but also the maximum size.

- **String (character)** – One or more characters, but now allowing spaces (unlike in the case of naming variables).

- **Complex** – MATLAB can also handle complex numbers. We will not concern ourselves with this further in this course however.

- **Object** – (We will not worry about *objects*, which can incorporate a combination of types. At least, not yet ...)

The first thing to learn is to ideally, not to attempt to mix up (combine) variables of different types. MATLAB is very forgiving when it comes to combining an integer and a real number in the same calculation, but in other programming languages, this should be avoided.

The second thing is how to assign a value to a variable (and in fact, create the variable in the first place). Programming languages such as FORTRAN require you to define the variable beforehand and assign it a type. MATLAB allows you to define and assign a value to a variable all at the same time, and it will kindly work out the correct type based on the value you assign to it. You assign a value using the assignment operator `=`[7]. For example:

```
A = 10
```

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

```
A =
10
```

Note that you do not need to add a space before and/or after the assignment operator (`=`). This is something of a personal programming and aesthetics preference, whether to pad things out with spaces or not.

MATLAB will also report in the *Workspace* window, the name and value, type (called *Class*), etc of all your current variables (just one currently?). Actually, it is not all quite so simple of you look at the *Class* of the variable `A` – it is *double* 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[8].

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

```
B = apple
```

and MATLAB is far from happy. Remember, that a string can be a variable name (or function), and this is what MATLAB looks for (i.e. a match to `apple` in the list or variable (and function) names). To delineate `apple` as a string, you need to encase it in (single) quotation marks:

```
B = 'apple'
```

Just as MATLAB creates new variables on the fly, you can reassigned 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** for the variable `A`. This is reflect in the updated variable list details given in the *Workspace* window.

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

[8] If you genuinely wanted an integer, there are ways to do this.

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

```
A = 'banana';
```

1.2.2 *Numerical expressions*

You can do normal maths in MATLAB. Or at least, something that looks at least somewhat intuitive.

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

The order in which the numerical operators are written down is important and MATLAB will execute them in a specific order (operators executed first at the top):

```
^
*,/
+,-
```

There is also 'negation', when you change the sign of a variable, which comes immediately after exponentiation. The assignment operator (`=`) comes last. If you are unclear about the order numerical operators are carried out, then place parentheses `()` around the component of the calculation you wish to be carried out first. For example, consider:

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

1.2.3 *Relational and logical operators*

We will see more of *relational and logical operators* later. 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 hence tests the relationship between 2 variables. For now, note in particular that the `equality` symbol is TWO = characters, and remember that a single = character is the `assignment operator`.

The outcome of this test is a 'yes' or a 'no'. Except in MATLAB (and other languages), the answer is given as the binary (logical) equivalent where 'yes' is represented by `1` and 'no' by `0`.

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

- `or` – symbol ||.
- `and` – symbol &&.
- `not` – symbol ~.

1.2.4 *Functions (built-in)*

MATLAB provides numerous built-in `functions`[9]. These `functions` are assigned names and so care needs to be take not to give a variable the same name as a function to avoid getting confused further down the road. Giving an exhaustive list (and brief description) is outside the scope of this document[10]. Common `functions` will be progressively introduced as the Chapters progress however. 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.

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

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

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

```
» pi
ans =
3.1416
```

In this example, the function is rather trivial – you need to tell the `pi` absolutely nothing, and it spits back the same things each and every time. In most other functions, you will pass some information, and the return value will depend on the input.

1.2.5 *Miscellaneous commands*

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

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

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

[ADD: convert number to string]

1.3 *Vectors and arrays*

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

1.3.1 *Vectors*

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

```
B = [0.0 1.0 1.5 2.0 2.5]
```

Values are extracted by specifying the index of the element required (counting along, left-to-right), e.g.

```
» B(5)
ans =
2.500000000000000
```

Equally, you can replace an existing value by assigning the new value to the appropriate indexed position. e.g. to replace the first element with a value of `0.5`:

```
B(1) = 0.5
```

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
1.0000
1.5000
2.0000
2.5000
```

1.3.2 *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[11]:

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

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

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

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

Elements in the matrix can be addressed using the syntax:

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

where `i` is the row number, and `j` is the column number. (It is very very easy to keep forgetting in which order the rows and columns are indexed. 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).

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.

1.3.3 *Matrix manipulation*

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

```
» 2*B
ans =
0 2 3 4 5
```

and

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

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

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

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

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

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

This is where the `transpose` function (`'`) comes in handy. It flips a (2D) matrix around its leading diagonal (columns become rows, and rows, columns)[12] .

```
» C'
ans =
255 221 357 85
```

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.'

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

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

```
119 17 289 255
187 102 85 340
102 170 51 153
```

(transposing the matrix turns the rows into columns)

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

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

Finally, you could transpose the answer:

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

now with a row vector gives you a format that looks like the row sums of the original matrix `C`.[13]

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

1.3.4 *Matrix math*

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

1.4 *Loading and Saving*

1.4.1 *Loading and importing 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 go through a couple of (hopefully!) slightly-less tedious data-based examples. If nothing else, you might accidently learn some Science even if nothing much about MATLAB ...

For the first example, to illustrate loading and importing of data as well as some basic array manipulation, we are going to transform a sediment core $\delta^{18}O$ time-series into an estimated history of glacial-interglacial changes in sea-level. The backstory is ...

Throughout the late Neogene[14], sea level has risen and fallen as continental ice sheets have waned and waxed. The main cause of sea-level change has been variation in the total volume of continental ice and resulting change in the fraction of the Earth surface H_2O

[14] 23.03 millions years ago (end of the Oligocene) to present is the Neogene Period in Earth history.

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

'101' – Naturally occurring oxygen is mostly ^{16}O, with some ^{18}O. ^{18}O is heavier, than thus water containing ^{18}O evaporates very slightly slower than $H_2{}^{16}O$. Water vapour, rainfall, and thus snow and ice, therefore has a smaller fraction of ^{18}O compared to ^{16}O than the sea-water that is left behind. More ice on land therefore equals more ^{18}O enriched sea-water.

Glacial ice has a lower $^{18}O/^{16}O$ isotopic ratio than seawater. When ice volume is high, seawater has relatively high $^{18}O/^{16}O$ ratio. When ice volume is low, seawater has relatively low $^{18}O/^{16}O$ ratio. If the average $^{18}O/^{16}O$ ratio of glacial ice is constant with time, then the average $^{18}O/^{16}O$ ratio of seawater approximates a linear function of the total volume of ice. Because changes in $^{18}O/^{16}O$ reflect changes in global ice volume, they also reflect changes in sea-level.

Foraminifera are little zooplankton bugs sitting on the ocean floor waiting for a meal to fall on top of them or swimming around in the surface ocean looking to make something into a meal, and make shells out of calcium carbonate ($CaCO_3$) – the stuff of limestones (although limestone is usually coral hard parts) and chalks (although chalks are usual calcitic phytoplankton rather than zooplankton shells).

The $^{18}O/^{16}O$ ratio of foraminiferal calcite isolated from marine sediments is primarily a function of the $^{18}O/^{16}O$ ratio of the water together with the temperature of the water. However, we will not concern ourselves with temperature corrections here but instead assume that foraminiferal calcite $\delta^{18}O$ only reflect changes in (global) ice volume and sea-level.

By measuring the $^{18}O/^{16}O$ value of calcite down-core we are sampling $^{18}O/^{16}O$ with a progressively older age. In this way we can reconstruct how ocean $^{18}O/^{16}O$ has changed over time. These measurements are reported in units of parts per thousand (o/oo) and written as '$\delta^{18}O$'. For this tutorial, don't worry about what exactly it means or how it is derived.

So, say that I want to run a global vegetation model for the penultimate ('second-to-last') glacial maximum (i.e., at around 141 ka (thousand years ago)). But I need to know how much land area to allow plants to grow on. Because a lower sea-level means that some of the continental shelves currently under the sea will be exposed, there is more land area available during glacials than at present (although some of the area available today was covered by ice sheets back then). But just how much? Suppose that I know that I can calculate land area if only I knew the sea-level at the time ... Evidence from dated coral reef terraces suggest that sea-level was around 117 m lower at the peak of the last glacial (ca. 19 ka). But what about the glacial before that (at 141 ka), which is the one that I am interested in? Was it the same; lower, or higher? Unfortunately, we run out of

useful coral terrace data from this time (Figure 1)? Instead, we are going to use foraminiferal calcite $\delta^{18}O$ to extrapolate the sea-level change we do know (117 m at 19 ka) back in time (see figure 1.1).

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

- Load this file into MATLAB as follows. The command we are going to use is `load`. Go call up the relevant `Help` page [15] on this function to find out the correct syntax[16] in the use of this command[17] . Note that by default, MATLAB looks to a file directory located within its installation directory (`$MATLAB/data`). So, where the `load` command requires a filename to be passed, you will need to enter either the full location of the file; i.e., starting with the drive letter (e.g. as per displayed in the Windows filemanger address bar, or the relative path to where the file is located (e.g. if there is a subdirectory called data, you will pass `data/sediment_core_d18O.txt`[18]. Alternatively, you can change the MATLAB directory that you are working in. (This works similar to UNIX/LINUX for those of you who are familiar with navigating your way around these operating systems.) You can make the download directory the default directory for working from by typing:

```
» cd DIRECTORY_PATH
```

where `DIRECTORY_PATH` is the path to the data directory[19], remembering that `DIRECTORY_PATH` is a string (i.e. enclosed in '). Or ... you can add a 'search path' so that MATLAB knows where to look. (Note that both these alternative possibilities can be implemented from the GUI.)

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

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

  1. At the Command line (»), type in the array name.

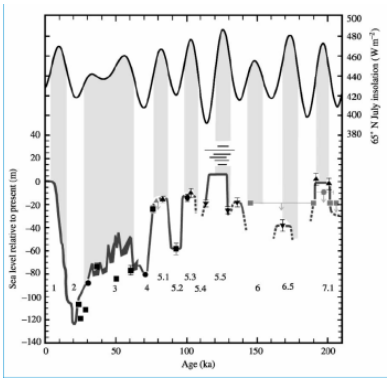


Figure 1.1: Direct sea-level data (210 kyr to present) from corals (solid symbols) and speleothems (gray symbols). July summer solar insolation at 65 degrees north latitude is depicted in the top panel. This is thought to control the glacial-interglacial variations in the size of the Norther Hemisphere ice sheets. The vertical gray bars indicate times of high insolation and the correspondence with minima in ice volume (equivalent to maxima in sea-level). (from; Holland, H. D and K. K. Turekian, eds, Treatise on Geochemistry, Elsevier, 2004)

[15] `» help load`

[16] The details of any required sets of parentheses or brackets, passed parameters, punctuation, etc etc.

[17] In this example, the file format is simple plain text, or 'ASCII'. If you view the contents of the file in a text editor such as Windows **notepad**, you will see that the lines of heading information in the file start with a `%` (the MATALB `comment` character). This tells MATLAB to ignore these lines (and hence jump straight to the 2 columns of data. The `load` function is not very powerful or flexible and we'll see some other ways of importing data later.
In addition, to use the `load` command, (i) the same number of values must appear on each line (row), and (ii) the values on each line must be numbers (reals or integers) and not strings. The file delimiter (character between each element in a row) can be a blank, comma, semicolon, or tab.

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

[19] You can view the files that are present in the directory that you are working in by typing (more LINUX-speak): `ls`. The command `addpath` will add a search path to the MATLAB workspace. e.g.

```
addpath DIRECTORY_PATH
```

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

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

3. At the command line, list the contents of the array again to view the change you have at the end of the data-set. Use the **up arrow** to bring up the command you want rather than typing it in again. Now delete this new row . Note that it is easy to get confused with which row number you need to address – although the data starts from year 0, MATLAB always counts the index (the sequential integer counting of the row or column number) of a location from 1 (one). (So age 10 ka is on line 11, and age 200 on line 201, etc.)

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

- Having reconstructed the sea-level for the past 782 thousand years, maybe we would like to find out some things about average conditions on Earth and how things have varied. Work out the following:

  The mean sea-level between 0 and 782 ka[20].

- So far everything has been in δ^{18}O units and time as kyr. As a warm-up, convert the units of time to years, i.e. multiple the first column of the data array, by 1000.0.

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

  Scale the δ^{18}O so that it represents changes in sealevel, relative to modern (0 m)[21].

- It is interesting to find out what is highest sea-level that we would predict – there are proxies for paleo-shorelines and coral reef terraces that could be used to validate (ground-truth or test) our simple δ^{18}O-scaled sea-level model. Determine the maximum

[20] HINT – try:
`» help mean`
(and which at the bottom of the help will provide you with a list of other, related functions, under `See also ...`)

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

and minimum sea-levels that have occurred over the last 782,000 years (easy if you have been following properly earlier ...). You could just go through looking for the highest and lowest values, but this is not very exciting with only 783 data points. If you had 10,000s of data points, doing stuff by hand is clearly going to use up all of your beer time.

Related to this, it would be helpful to know *when* the minimum and maximum sea-level heights occurred. This is going to involve using the `find` function, to find the data row in which the minimum and maximum values have occurred. Once you know the respective data rows, you can then easily pull out the ages.[22] Find the ages of both minimum and maximum values.

[22] HINT – if your maximum value was stored in the variable max_value, you found find the corresponding row by:
`find(data(:,2) == max_value)`
What this is saying, is search the 2nd column (the sea-level values) of the array `data`, and look for a match to the value of `max_value`. The equality operator (==) is used in this context.

For the second example, to illustrate loading and importing of data as well as some basic array manipulation, we are going to process a paleo atmospheric CO_2 proxy dataset. The file is called `paleo_CO2_data.dat` and is as before, available from the course webpage. Open the file up (view it) with a text editor (e.g. Windows **notepad**) and note the format – there are a bunch of header lines and moreover, some of the columns are not numbers (but rather strings). So even if you were to edit out the headers with `comments` (`%`), you are still left with the problem of mis-matched columns. You could edit the file in **Excel** to remove the problematic columns ... but now this seems like a real waste of time to be editing data formats with one software package just to get it into a second! (Again, you could use the MATLAB GUI import functionality ... but it will be a healthy life experience for you to do it at the command line :o))

To start with, you are going to need to know the format of the data in advance. (You have determined this already by viewing in a text editor.) Then following these steps:

1. First 'open' the file – you will be using the function command `fopen`, and passing it the filename[23] (including the path to the file if necessary). So that you can easily refer to the file that you have opened later, assign the output of `fopen`[24] to a variable, e.g.

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

2. This is where it gets a little tricky ... the function you are going to use now is called `textscan`. Refer to the `help` function 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

[23] For convenience, you could assign the filename (+path) to a (string) variable and then simply pass the variable name (no ' needed).

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

According to MATLAB `help`:
"the `format` is a string of conversion specifiers enclosed in single quotation marks. The number of specifiers determines the number of cells in the cell array C." Take this to mean that you need one format specifier, per column. 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)integers
%f - floating point numbers
%s - strings
```

gets really miserable, but hang in there!) You simply list, space-separated, and between a single set of quotation marks, one format option per element of data.

(c) A parameter together with an (integer) value[25], to specify how many rows of the file, assumed to be the header information, to skip.

[25] The format is of a parameter name, in '', followed by (after a comma) an integer for the number of lines to skip. The parameter name is `Headerlines`.

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

An example is needed here to highlight the path out of the MATLAB syntax mire ... We are going to load in just the first 2 columns of data which are both integers in this case, and skipping the first 3 lines of the file. Skipping the first 3 lines is the easy bit (we pass: `'Headerlines', 3`). Omitting all the information following the first 2 elements is trickier and something for Google to help in[26].

[26] 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]`).

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

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

```
» fclose(openfile_id);
```

3. Actually, it does get worse before the end of the tunnel ... what `textscan` actually returns, i.e. your previous read-in data, is placed into an odd structure call a `cell array`. It is not worth our while worrying about just what the heck this is, and if you view it in the **Variables** window (i.e. double click on the `cell array` name in the **Workspace** window), it does not display the simple table of 2 columns of data you maybe 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 = call2mat(my_data);
```

And now ... it is done, i.e. there exists a simple array, of 2 columns, the first being the age (Ma) and the second the CO_2 concentration value (units of ppm). :)

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

We can now actually do something with this data. Perhaps, as a common way of displaying paleo atmospheric CO_2 and O_2 concentrations is as a concentration unit relative to present-day[27]. All we need to do, is divide the paleo CO_2 concentration data by the modern value. However to make it more 'fun', lets divide the data by the value in the CO_2 concentration dataset closest to modern (i.e. 0 Ma).

[27] known as 'PAL' – Present Atmospheric Level

So your task is to firstly find[28] the data entry with an age closest to zero, determine the CO_2 concentration corresponding to this age, and divide everything (the CO_2 concentrations) by it.

[28] CLUE: you are going to use the `find` function ...

1.4.2 *Saving and exporting data*

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

There is also an equivalent `textscan` also involving explicitly opening and closing files called `fprintf`.

1.4.3 *Loading and saving the workspace*

The entire workspace (including all variables and their values) can be saved to a file and then later re-opened. The file format is specific to the MATLAB program and the file-name extension by default is `.mat`. This might prove helpful in large modelling projects and particularly if you do not come back to work at the exact same computer each time.

2
Plotting and visualizing data

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**.

2.1 *Introduction to graphics and figures in MATLAB*

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

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

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

So, you need to pass it your x-axis data vector[3], followed by your y-axis data vector (comma separated). Do this, and depending on just what or how random your y-axis data was, your should end up with something like Figure 2.1 in a window captioned "`Figure 1`".[4]

This ... is easily the least professional plot ever. And one that breaks all the most basic rules of scientific presentation, such as labelling axes (there is also no title, although here I have added a figure caption in the document and so I can get away with it). But this is the default state and you'll need to do a little more work on it. Note that by default, MATLAB scales both axes to closely match the range of values. In the example here, the default min and max axes limits are in fact the min and max values in the x and y-axis data.

[1] Refer to the first chapter and the subsection on matrices if you have forgotten how.

[2] If you find that you have created 2 rows of data rather than 2 columns, remember that you can swap the rows and columns with the `transpose` function.

> You can 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).

[3] Again – don't forget the previous lesson where selecting a specific column of a matrix, which is a vector, was covered.

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

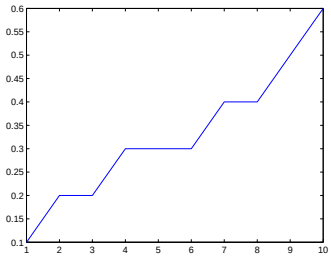


Figure 2.1: Default output of `plot`.

However, if the maximum y value was vary slightly larger, you'd see that MATLAB would adjust the maximum y-axis limit to the next convenient value so as to preserve a relatively simple series of labelled tick marks in the axis scale. In fact, try that. Replace your maximum data value, with a value that is very slightly larger.[5] 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).

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

[5] 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. Or better: use the `max` function and then `find`, as per the previous tutorial.

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

When MATLAB displays text, be aware that there are a bunch of special characters that may not come out as the character you want. The more common ones are:
`_ - will make the following character a subscript, or a sequence of characters if you place them within a pair of curly brackets {}.`
`^ - will make the following character a subscript, or a sequence of characters if you place them within a pair of curly brackets {}.`

As an example, load the oxygen isotope data from the previous tutorial and convert it into sealevel. Using the command line only ... plot it, label the axes, add a title, and set the age scale from 0 to 800 kyr, and the sealevel scale from -120 to +20 m (relative to present-day). At this point it should look something like figure 2.2. (I cheated a little here and changed the font sizes by passing an additional pair of parameters to the axis label and title commands, of the form: `'FontSize',SIZE`, where `SIZE` is the font size (in units of points (pts)), e.g. 18 for the title and 15 for the axis labels in this example.)

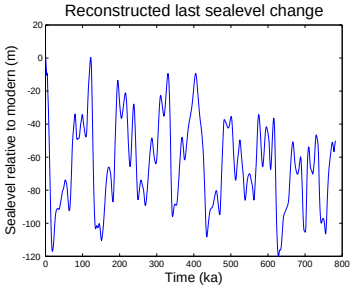


Figure 2.2: Past sealevel variability as reconstructed from oxygen isotopes.

2.1.1 *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 [6]. 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).

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

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

2.2 *Fancier 1D plotting*

2.2.1 *Modifying lines/symbols in `plot`*

We'll work though an example of some slightly more involved plotting of traditional 1D (i.e. y values against x) data. To begin, you be loading in a simplified version of the Phanerozoic CO_2 dataset (see previous tutorial). As `paleo_CO2_data.txt` you can just import it into MATLAB using the `load` function. However, unlike the marine sediment core oxygen isotope dataset, you now have 4 columns in the array[7]. The first column is age (Ma), the second the mean CO_2 value, and the 3d and 4th are the low and high, respectively, uncertainty limits. To start off: plot the mean paleo CO_2 value as a function of age (in Ma). If you closed the previous Figure window (see earlier), it is not essential to explicitly open one – 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 2.3. O dear ...

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

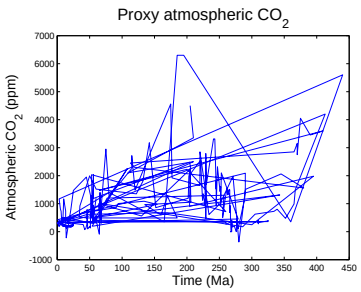


Figure 2.3: proxy reconstructed past variability in atmospheric CO2.

So ... that was not so successful. What is happening in the default behaviour of `plot`, is that each data point is being joined to the previous one with a line. This was fine for the sealevel example dataset because time progressed towards older and older values in the first column, e.g. the data was ordered as a function of age (the progression of age values in the case is called *monotonic*). If you view the CO_2 data, this is not the case. (In fact, in the original, full version of the data, ordering is by proxy type first, and then study citation, and only then age ...). Your options are then:

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

2. You could sort it in **MATLAB**. How? Well, a reasonable gamble, which actually turns out to be a win, is to try:

```
» help sort
```

Well almost. 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. As a `see also` MATLAB suggests `sortrows`. The help text on this looks a little more promising. It is still slightly opaque, so the best thing to do is to try it (and view the results)! This looks rather better. The resulting of `plot`-ting this is 2.4. (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 values in the Cretaceous and Jurassic and then lower again in the Carboniferous (roughly matching the progression of ice and hot house (and then back to recent ice ages) climates).

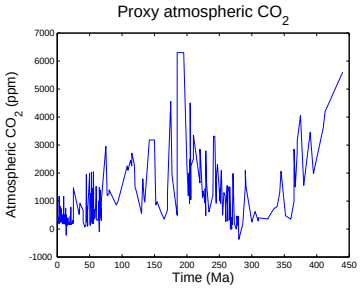


Figure 2.4: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

3. Or you could plot 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 ...

   The 3rd solution then is to specify a symbol and no line. The choice of colors is your oyster, as they (almost don't) say. e.g. Figure 2.5.

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.

2.2.2 *Plotting multiple datasets and multiple plot panels*

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

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

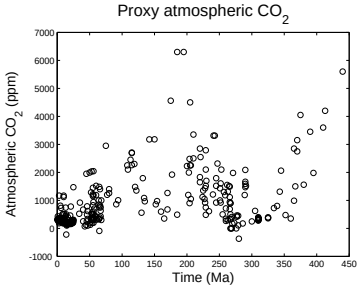


Figure 2.5: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

be `on`, or `off`.

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

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

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

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

and it should end up looking like Figure 2.6.

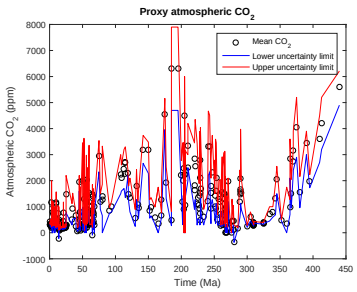


Figure 2.6: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

As a different example, create some sine waves using the `sin` function over the range $0 < x < 2\pi$:

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

Now create a graph with both plotted. Play about plotting with different symbols and/or line styles and different colors. Add axis labels, a title, and a legend.

Rather than plotting multiple data in the same plotting panel, it is possible to place several different plots on the same figure Ű this is done through the `subplot` command [8]. 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 2.7).

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,-y2,'r');
» subplot(2,2,4);
» plot(x,-y,'r');
```

[8] » help subplot

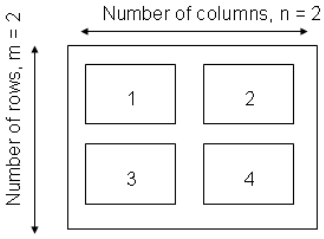


Figure 2.7: Arrangement of subplots.

2.2.3 *Scatter plots*

Back to the previous (CO_2) data, but a different spin on it. We'll start with the simplified CO_2) dataset that you have already loaded in (`paleo_CO2_data.txt`) (unless you have `clear`-ed the variable list) but in due course, you'll need the more complicated version processed in the 1st tutorial.

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

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

(where `X` and `Y` are vectors). This almost could not be more straightforward. Make yourself an `X` and `Y` vector out of the loaded-in dataset (or if you are feeling brave, you can pass in directly the appropriate parts of the dataset array), close the existing Figure window[9], and `scatter`-plot the (mean) CO_2 data.

Perhaps a little disappointingly, the default (Figure 2.8) (plus added labels) looks a little like one of the plots before. However, `scatter` can plot color-filled symbols, but more powerfully, can scale the fill color to a 3rd data value (vector), in a sort of pseudo 3D *x*-*y*-*z* plot. For instance, it will be duplicating information that is already presented (*y*-axis), but you could color-code the points, by the y-value, i.e. the atmospheric CO_2 value. e.g.

```
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 2.9

There are a number of variants[10] on this theme. For instance, you could scale the point size by a data value. We could do with by:

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

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

This leads us to a refresher on the use of `find` and some basic data filtering. The simplest thing you could do to ensure no zero values, would be to add a very small number to all the values. This would defeat the 'no zero' parameter restriction, but would not help if there were negative values and you have slightly modified and distorted the data which is not very scientific. Substituting a `NaN` for

[9] See earlier.

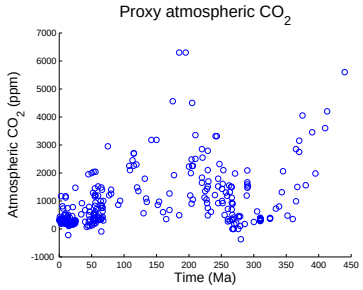


Figure 2.8: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

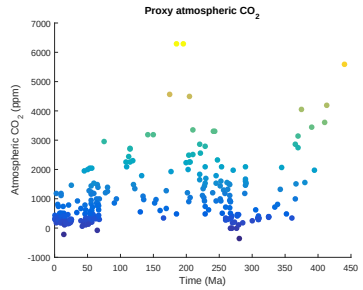


Figure 2.9: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

[10] As always, refer to:
» `help scatter`

problem values is a useful trick, as MATLAB will simply ignore and not attempt to plot such values. So first, lets replace any zero in the CO_2 column of the data with a `NaN`. The command you would use is:

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

As ever – 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;
```

This saying: find all the locations (rows) in the 2nd column of `data` which are equivalent (==) to zero. Set the CO_2 value in all these rows, to a `NaN` (technically speaking: assign a value of `NaN` to these locations). You have now filtered out zeros, and replaced the offending values with a `NaN`. Alternatively, we could have simply deleted the entire row containing each offending zero. Breaking it down, this is similar to before in that you start by identifying the row numbers of were zeros appear in the 2nd column, but now we set the entire row to be 'empty', represented by `[]`:

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

If you check the **Workspace window**[11], 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. It turns out that there is a CO_2 proxy data value less than zero(!!!) We can filter this out, just as for zeros. I'll leave this as an exercise for you ... If ('when', of course!) successful, the plot should look like Figure 2.10. As another lessonette, given that the circles are insanely large ... try plotting this with proportionally smaller circles[12].

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

It turns out this is not entirely trivial because as you have seen, the data is not as well behaved as you might have hoped. In fact, it

[11] Or:
```
» size(data)
```

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

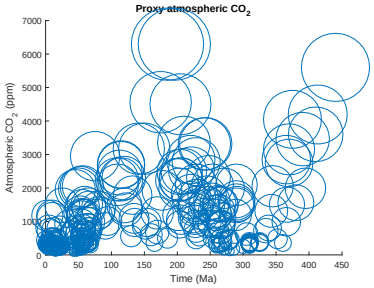


Figure 2.10: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

is just like real data you might encounter all the time! Before you do anything – break down into small steps what you need to do with the data, as this will inform what (if any) additional processing you might have to carry out on the data. It should be obvious, that to create a CO_2 difference, *relative* to the mean, you are going to have to divide by the mean value (column #2 in the array). So first off – if any of the mean values are zero, it is all going to go pear-shaped. Actually, equally unhelpful, or at ;east, lacking in any meaning, may be negative values. If you inspect the data (in the **Variable window**), there are both zeros and negative values for mean CO_2 proxy estimates. We need to get rid of these. Follow the steps as before. You may also have to process the min and max values should they turn out to be the same. Likely you are going to have to delete all the rows in which (1) column #2 values are zero or below, and (2) column #3 and #4 values are equal (you could also try the `NaN` substitution and see if it works out). (If you need a slight hint ... one possible answer is here [13] , but try and work it out for yourself.)

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

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

It should end up looking something like Figure 2.11 in which you can see the high relative uncertainty (bight colors) prevail at low CO_2 values and 'deeper time' (ca. 200-300 Ma). The colorbar title (label) is maybe not ideal, nicer would be one aligned vertically rather than horizontally. We'll worry about that sort of refinement another time.

One final example in this section to introduce some new plotting functions, but also to quickly go back over some basic array manipulation and processing. The data we will be analysing have been taken from the USGS. The quake data are extracted between -5 and 20 lat, and between 90 and 105 lon, starting Dec 26, 2004 and ending June 30, 2005. The data file can be found on the course webpage (`data_USGS.txt`). The columns are: (1) day, (2) latitude, (3) longitude, (4) depth, and (5) magnitude. Carry out the following:

[13]
```
» data=load('paleo_CO2_data.txt',
...'-ascii');
» data(find(data(:,2)<=0),:)=[];
» data(find(data(:,3)==data(:,4)),:)
...=[];
» scatter(data(:,1),data(:,2),40,
...100*(data(:,4)-data(:,3))./data(:,2),
...'filled');
» xlabel('Time (Ma)')
» ylabel('Atmospheric CO_2 (ppm)')
» title('Proxy atmospheric CO_2')
```

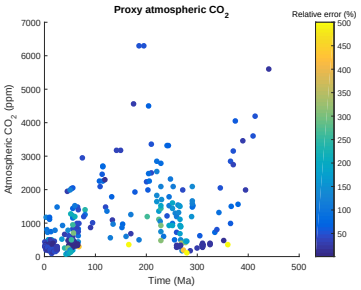


Figure 2.11: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

1. The first earthquake in the list is the Sumatra earthquake of December 26, 2004. The magnitude of this earthquake has been revised upward since the data was downloaded. Actually, most energy released in large earthquakes is in very low frequency shaking that most seismometers do not record. The real magnitude had to come from a special analysis of "normal modes", or standing waves on the Earth's surface with periods of up to 54 minutes! When the media said that the Sumatra earthquake made the Earth ring like a gong, these are the waves they were talking about. So since we know that the magnitude was really 9.3, first off, replace the value of the magnitude of the first earthquake in the array.
2. Identify the smallest magnitude of recorded earthquake. You should find that the minimum earthquake size on this list is 3.5. For an earthquake in California, the minimum magnitude would be more like 1. This is because this particular seismograph network did not have many instruments around Sumatra. Another problem is that the earthquakes are offshore. If the nearest seismograph is far from a small earthquake, that earthquakes may not be detected. This means that the data are artificially truncated. Since everything below 3.5 is missing, some of the M=3.5 to 4 earthquakes may have been missed, too.
3. Identify the minimum and maximum earthquake depths. The really deep ones (>40 km) are probably in the subducted slab that goes beneath Sumatra. The zero depth means that it could not be resolved - most hypocentres are 4 km or deeper. (hypocentre = like epicentre, but at depth: the point on the fault where the earthquake rupture starts)
4. How many earth quakes in total were recorded?[14] A little harder: How may earth quakes were there bigger than M = 8? Determine how many quakes occurred bigger than M = 7, 6, 5, 4, and 3. Determine the day on which the magnitude 8.7 shock occurred.

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

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

There is only just so much looking at and processing raw data you can do before your eyes start ... to ... ... droop ... ... and ... ... Zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz. OK – so now to visualize what is going on. Plot using the scatter function the locations of all the quakes from day 0 to day 91 (inclusive), and in a second plot the locations from day 92 onwards.[15] The first area covers the area that ruptured in the M 9.3 quake (1200 km long and 100 km wide) and the second, to the South, is smaller. This is important because the aftershock distribution made people very wary of the (low) early magnitude estimates - the area of dense aftershocks often delineates

[15] Note that to find the position (i.e., row number) in the array at which day 92 data starts, you can either do it the dumb way and scroll through the **Variables window** like some sort of **Excel** Muppet, or do something more clever:

HINT: find the array locations with day > 91, but then you want to find the minimum value of the long list of row locations that **MATLAB** will spit out that satisfies this criteria.

the part of the fault that ruptured, and scaling laws relate rupture length to magnitude.

Create a figure with multiple panels, showing:

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

2.2.4 *Histograms*

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

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

```
» hist(data_USGS(:,5),M);
```

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

3. Now plot the histogram of quake frequency against time (i.e., day number) up to day number 186. You will have to assign a new vector of values to `M`, one that starts at 0.5 and ends at 185.5. Omori's Law says that the number of aftershocks per day should decrease following a power law – does this look to be the case (approximately)? (One problem is that the small earthquakes are missing which makes it appear not to work so well!)
4. Try this again (i.e., frequency of quakes vs. time), but investigate the effect of changing the bin size – try making the bins about 1 month (30 days) in duration. Note that now M must start at 15.0 (the mid-point of the first monthly bin). Sometimes changing the

bin size can help if the data is noisy, but sometimes you lose important information. Which was better do you think – can you still see a power-law decay in quake frequency following each major event with the data in monthly bins? If you want, experiment with other bin sizes to see how the data comes out. There is not always a 'right' answer in plotting data and sometimes you just have to experiment a little to see what looks good.

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

2.3 *x,y,z (spatial) plotting*

One could regard the previous scatter plotting as a sort of *x,y,z* plotting, in as much as a 3rd dimension (*z* data value) was represented through color and/or symbol shape, although primarily it was an *x,y* plot with some fancy extra options. However, MATLAB provides a wide variety of more formal ways of plotting data spatially, either in 2D (*x,y,z*) or even with a 3rd spatial dimension and a 4th dimension representing the data value (*x,y,z,zz*) (see Box).

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.

x,y,z PLOTTING

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

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

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

As an example, load in the (simple format) bathymetric data file (**etopo1deg.dat**) from the course webpage. This is the height of the (solid) surface of the Earth relative to mean sealevel in meters, with the continents having a positive value and the ocean floor, negative. The data is conveniently on a 1° (longitude and latitude) grid. You could view the resulting elements of the 2D array in the Variable window if you like ... but at 360×180 in size, there may not be much of use you can glean by visually inspecting the matrix[16].

Try throwing the array into the `image` function see what happens (hopefully something like Figure 2.12). It it had happened to come out displayed upsidedown, then you'd need to flip the matrix upsidedown using the command:

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

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

```
etopo1deg=etopo1deg';
```

**** basic spatial plotting, the `image` function ****

[16] 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.

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

This is not exactly the prettiest of images. You can distinguish ocean (blue) from land (mostly brown, but other color pixels in places). Fortunately, MALTAB provides a variant of this plotting function, `imagesc`, that calculates the color scale to exactly span the min/max values in the data. Try it (and get something like Figure 2.13 hopefully).

The function `imagesc` also enables the range of data values the colo9r range corresponds to, to be set. Refer to `help` on this function and see if you can plot just the above-sealevel, i.e. land surface heights, spanning zero (sealevel) to the maximum height[17].

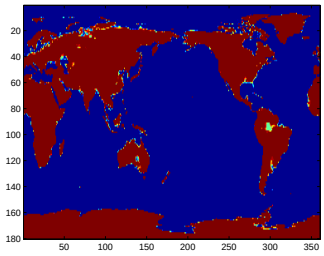


Figure 2.12: Very basic imaging (`image`) of an array (2D) of data – here, global bathymetry.

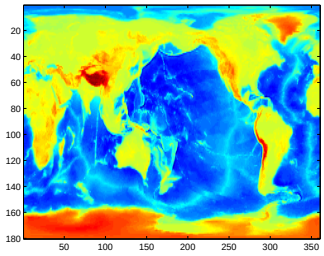


Figure 2.13: Slightly improved very basic imaging (`imagesc`) of bathymetry data.

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

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

It would be nice to be able to plot this temperature data for ourselves and have more control over its presentation and hence the message you are trying to convey with it. Pick one (any one) of the the monthly global surface temperature data-files on the course webpage and download it.

Pick one of the **MATLAB** 3D plotting functions (see Box). Your choice. Read the relevant `help` for your chosen *function*. Plot the dataset using the simplest usage of the *function*.

You should have got 'something' (see Figure 2.14). But you'll note (hopefully) that you haven't got any grid information yet; i.e., you don't know what the longitude and latitude axes should be and the default graduation clearly cannot be for a planet. You could guess that latitude (y-axis) goes from 90°S (-90°N) to 90°N, but be careful – sometimes you will see global maps only plotted between say 60°N or 75°N if the highest latitudes are not very interesting or e.g. it is a satellite product and the satellite cannot observe high latitudes. You would be safer guessing that there is likely to be the full 360°of longitude; but starting where? Common longitudes to start plotting from in the literature are 0°E and -180°E (180°W). The second thing to note, apart from missing x- and y-axis labels (and title) that you could

**** (x,y,z) (`contour/contourf`) plotting, the `meshgrid` function ****

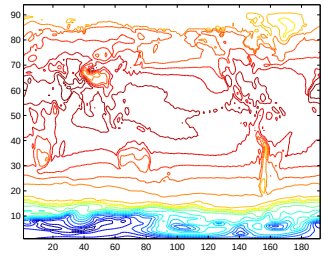


Figure 2.14: **Example contour plot.** Result of `contour(data,20)`, where the data file was `temp7.tsv`.

easily add in, is; what do the colours mean? i.e., If you were asked what the temperature was over the Equatorial Pacific, or what the coldest temperatures in the northern Arctic were; well, what do the red and dark blue colours actually mean? 0°C, 50°C, 100°C??? Your contour or colour contour (depending on which 3D plot you prefer) is clearly missing critical information and incomplete.

OK – we'll fix the (lon,lat) information to start with. If the (lon,lat) grid information was available from the same website where the temperature distribution data came from, then you already know how to use this (e.g. from `help`) when calling the contour (or `confourf`) function; i.e.

```
» contour(lon,lat,data)
```

instead of just supplying the array of data values on its own, i.e.;

```
» contour(data)
```

(and similar to Figure 2.14).

But suppose the actual data-files (arrays) of (lon,lat) information are not available at all. Anywhere. Then what? (In fact, this is the case with this particular on-line data repository.) The information that the website did provide (click on the 'data in view' button on the webpage you were looking at) is that:

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.

You could create a vector (similar to as per you have done previously) to try and solve this, by entering something like:

```
» lon = [0:1.875:358.125];
```

In fact, type this in at the command line and view what it gives - is this sufficient longitude information for the 94×192 temperature distribution data-set (94×192 is the array size of the temp global temperature distribution data which is displayed in the **Workspace window**)? Can you create the appropriate 94×192 array out of this single 1×192 vector? Maybe. But not easily.[18]

Helpfully, **MATLAB** provides a special function called `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 clean 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

[18] It turns out you could hasve easily ... but only if you had skipped ahead and gone through the next Tutorial ...

with the rows all having the same values. This is exactly what we need for defining the (lon,lat) of all the global surface temperature distribution data points.

As an example of this, suppose you wanted to create the (lon,lat) grid information for a data-set covering the LA and the Inland Empire, that went from; -120°to -116°(East), and 32°to 36°(North). The arrays we will use to store the longitude and latitude information in we will call; lon and lat. Having looked at `help`, the command you will issue hopefully is apparent:

```
» [lon lat] = meshgrid(-120:1:-116, 32:1:36);
```

A translation of this is: create a pair of matrixes, one for lon values going from -120 to -116 with a step size of one, and one for latitude values going from 32 to 36 with a step size of one, and assign them to a pair of variables, `[lon lat]`. Type this in at the command line and view the contents of the `lon` and `lat` arrays to convince yourself that it actually works out.[19]

Now go create a suitable pair of (lon,lat) arrays for the global temperature data. You will need to remember to use the `colon operator` to increment longitude in 1.875°steps and latitude in 1.904°steps. You may as well call the arrays that you create `lon` and `lat`. Have a look at the arrays that you have created (in the **Variables windows**), and satisfy yourself that for each and every temperature point in the temperature data array (i.e., (row,column) location), the corresponding locations in the lon and lat arrays gives the full (longtitude, latitude) location of the temperature value on the Earths surface.

Plot the global temperature distribution on its proper (lon,lat) grid. Go label the axes if you haven't already done this.

Now you need to fix the problem of not know what any of the colours (contours) in your beautiful plot mean ... Note that the temperature of the raw data-sets is in Kelvin (or did you really think that January temperatures in Socal were around 290°F?). Go change the dataset to some more sensible units – either a simple conversion to degrees Celsius, or Google how to convert to Fahrenheit. See if you can produce something like (or better than!) Figure 2.15.

For another `contourf` example – load the bathymetry data and plot it in solid color contours. The bathymetry data is on a regular 1 degree grid starting at 0° longitude and so the call to the `meshgrid` function will have to be adjusted accordingly. By creating and adjusting a vector to define the contour intervals, see if you can plot: (1) ocean floor and land in 1000 m intervals, (2) the land (only), in 500 m intervals, (3) the continental shelf exposed at the last glacial, which we'll call ocean floor shallower than 120 m water depth, in 5 m

[19] Note that the latitude numbers in the lat array count in the opposite direction (numbers getting larger going down the rows) to how you would expect if you were looking down at a map. Remember that arrays in **MATLAB** count from the top left (columns across from the left, and rows down from the top) rather than in a map, which is orientated from the bottom left. If at the end of the day when you plot your data you find that you get an up-side-down map, then you know that you need to simply just flip the lat array around. Refer to the earlier Tutorial and/or Look up `help flipdim` for one way of re-orientating the data in an array.

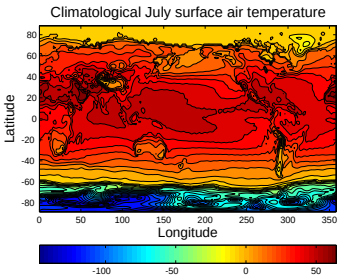


Figure 2.15: **Example contour plot.** Result of `contourf(lon,lat,temp7,30)`, where the data file was `temp7.tsv`, with some embellishments.
**** (x,y,z) (`contourf`) plotting, the `meshgrid` function ****

intervals.

2.3.1 Plotting maps

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

3
MATLAB scripting and 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.

3.1 Introduction to scripting in MATLAB

Commands in MATLAB can become very lengthy, and you might end up with a lot of different lines of stuff 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 of things. And all the while, the clock on the wall of the bar is ticking. Tick tock, tick tock, tick tock. (The clock is also tocking.) ... If only there was some way of storing all these commands in such a way that they could be 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 execute them all (sequentially) all at one go. **MATLAB** gives this text file a fancy name (because it is a very fancy piece of software, after all) – a *script*[1], otherwise known as an `m-file`. To create a new `m-file`; from the File menu, select **Script** (a common type of `m-file`)[2]. You will see a text editor (more fancy-ness) appear in front of your very eyes, containing your requested (but currently empty) `m-file`. Save the `m-file` to your directory of choice. Alternatively, simply create a new (blank) text file and saving it with the extension `.m`, rather than e.g. `.txt`, creates you a (script) **m-file**. From an **m-file**, you can issue all the **MATLAB** commands you previously would have entered individually, line-by-tedious-line, at the command line. Furthermore, having created and saved a **MATLAB** script, it can be executed again and as many times

`m-file`
A simple text file, in which a series of one or more **MATLAB** commands are written and which via the .m extension, **MATLAB** interprets as (1) a program file that can be edited, and (2) *script* or *function* that can be 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`.

[1] The conception of a *function*, will be introduced later.
[2] In order version of **MATLAB**: **File/New** menu, and select: **Blank M-file**.

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**[3]. You can also run the *script* (`m-file`) by hitting the big bright green Run icon button at the top of the **m-file** editor[4]. The short-cut for running it is to whack your paw down on the Function Key **F5**.

A few tips about **MATLAB** scripting before we go on (and on and on and on):

- Choose helpful variable names so that it is clear what each variable represents, but avoiding *excessively* long names.
- Use comments within your `m-file` to add explanation and commentary on your program. Anything after a `%` on the same line is a considered a comment, 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*).
- Always save your changes before running your program (or you may unknowingly be running the previous version).
- There are quick ways to run just portions of your code;
  1. Highlight, and copy, and paste in the command window.
  2. Highlight some code, right click, and select **Evaluate Selection** (in the current **MATLAB** version, there is also an icon for **Run Section**).
  3. Or highlight and hit the **F9** key.
- If using the script to do some plotting, sometimes it is convenient to add at the top of the `m-file`,

```
close all;
```

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

[3] See previous Tutorials for comments on changing directory vs. adding paths.

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

Creating `help` text in an `m-file`

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

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

*** *scripts* ***

Your first 'program' ... inevitably ... has to be to print 'Hello World' to the screen. No, really. (Google it.) Create a new `m-file`, calling it e.g. `hello_world.m`. You need to use the function `disp`[5], which will print to the screen, either any text you specify (in inverted commas), or the value of a variable (which could also contain character information). For now, simply pass the text directly. Your program needs just a single line in the m-file:

[5] » `help disp` as always, for function syntax and usage.

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

Save the file. 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 ... ?)

*** use of m-files (scripts), use of the colon operator, code debugging ***

For an example , loosely based on the previous Tutorial – go create a new **m**-file called: **plot_some_dull_stuff.m**[6]. Then add the following lines to the file:

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

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

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

```
???  Undefined function or variable 'k'.
Error in ==> plot_some_dull_stuff at 11
plot(x,y2,k);
```

In this situation, the actual error reported might not always mean that much to you (it doesn't always mean that much to me, either). However, the line number at which the problem occurred is gold-dust. We know from the error reported by **MATLAB** that we have a bug in the code at line 11. See if you can de-bug the program. Look up `help plot` to remind yourself of the correct syntax on line 11 if it is not immediately obvious.

Once you have fixed the bug; save and re-run the script. Now you should see Figure 3.1.

Perhaps you find that you need to alter something fundamental in the script; perhaps you forgot that you advisor told you; 'Always plot from zero to 4 times pi, or you'll find yourself buried in a shallow grave.' Your advisor is 6'7" and 250 lbs with a mean temper, and is walking your way. Do you: (a) scream and run, or (b) simply edit the `m-file`, changing line 5 to `x = 0.0:0.1:4*pi`, before quickly saving (don't forget) the edited file and re-running the script (e.g., pressing F5). Your bacon is saved by using an `m-file`. If you had just tried to type in everything at the command line all over again, you might be dead by now ...

[6] Remember – you are advised to name your `m-files` as something vaguely descriptive of what the script actually does (and you do ont have to go with this choice, although it might turn out to be perfectly descriptive ;)

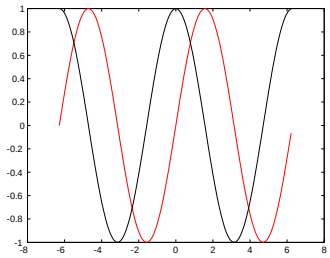


Figure 3.1: Output from the `plot_some_dull_stuff` **m-file**.

3.2 *Loops*

The first main program construct that you are going to see is the *loop*. There are a number of different forms of this in **MATLAB** (see **Loops** Box) (and also in other programming languages), but the basic premise is the same – a designated block of code (one of more lines of code[7]), 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).

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`.

Loops, camera, action! A humongous and ungainly example follows, in which we'll see the use of *loops*, recap some on loading in data files, plotting (and interpolating) 2D data, and see a few new tricks (aka **MATLAB** functionality). 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 start to appreciate the value of constructs such as *loops* in computer programming in saving you a whole bunch of effort and needless duplication of code.

So, first download all the monthly global surface temperature data-files on the course webpage (there are 12 files to download). Then you are going to want to plot them all[8]. This would get tedious if you had to do this at the command line 12 times. Think how much more of your life you would be wasting if we had weekly data. Or monthly data for 1972 through 2003, some 372 separate data-files ... You would never have time to drink beer ever again?

Create a new `m-file`. Call it ... anything you like[9]. However, as well as appropriately naming your script file, add a *comment* on the first line of the file as a reminder to yourself of what it is going to do. Also, for now, it is good practice[10] to use the commands: `clear all` and `close all`.

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

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

or

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

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

[7] 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.

**** the `for ... end` loop, string concatenation, comments, the `num2str` function, defining (color, contour) plotting scales, **MATLAB** `movies` ****

[8] Strictly speaking, you are going to be doing this, regardless of whether or not you actually 'want' to ;)

[9] `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.

[10] Note that there may be situations in which you want to run a script file to process some data that you have already loaded in – by issuing the command `clear all`, you will erase the **MATLAB** workspace and any data already loaded in.

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

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

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

What you have done here is to load the January 2D (lon-lat) temperature distribution into the 1st 2D layer of the temp array, and then we have gone and created a second 2D layer on top of the first with the February data in it.

Look at the **Workspace window** (or type `size(temp)`) – you now have a 3D (94×192×2) array. Fancy!

Go on and load in the March and April data in a similar fashion (check that your array now has dimensions of: 94x192x4). Are you getting bored yet? No? Then load in the May and June data.[11]

[11] Zzzzzzzzzzzzzzzzzzzz

You should be able to see a pattern forming here. This is something that a loop could be used for while you go off down the bar. We first need to construct the loop framework. We'll call the month number counter variable, `month`. Create a loop (with nothing in it yet) with `month` going from `1` to `12`.[12] Refer to the course text (this document!), and/or the 'Getting Started in MATLAB' booklet, 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[13]. What happens? Can you tell?

[12] 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 %.

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

One way of following what is going on as **MATLAB** executes the commands within a script is to explicitly request that it tells you how it is getting on. Use the function `disp` to help you follow what the program is doing. Within the loop, add the following line:

```
disp(month)
```

then save and re-run the script.

Now you can see how the loop progresses. This sort of thing can be useful in helping to *debug* a program – it allows you to follow a program's progress, and if the program (or **MATLAB** script) crashes, then at least you will know at 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[14] (unless you have grown rather attached to it) and move on.

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

We can construct filenames to load in by:

1. Converting the number value of a (count) variable to a string (`num2str`), and

2. forming a complete filename by concatenating other strings before and/or after this.

The `num2str` function is new to you – look it up in `help` for exactly what it does and the correct syntax. For the second part of this – recall that you can concatenate arrays (you have done this before with numbers in the arrays). The same can be done if your variables contain strings (i.e., a sequence of characters). For example, you can probably guess the outcome of (but type it in at the command line anyway):

```
» A = ['be' 'er']
```

Note that string information must go within inverted commas; ". We can do a similar trick to construct a long filename string. Type in the following example at the command line:

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

Now load (at the command line) the data file given by the filename string contained in the variable `filename`[15].

[15] HINT:
```
» load(filename);
```

You should see that you can construct filename strings from individual parts of strings (by concatenating), and you can pass the load command the array containing the filename – note that this is different to how you have been loading in data before when you have passed the actual string (which you have had to place between inverted commas to tell **MATLAB** it is a string). If you pass **MATLAB** a variable containing a string, then **MATLAB** will automatically look to see if the contents of the variable are a string or number, and if it wants a string input and your variable contains a string, **MATLAB** will be very happy indeed.

If you have to write out 12 times a line like `filename = ['temp' '1' '.tsv'];` then you still have not managed to save any drinking time. This is where you can use the loop count number (stored in the variable `month`) and convert this number to a *string* in order to automatically generate the correct month's filename each time you go around the loop.

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

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

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 (compared with 12 lines if you had to write it all out long-hand).

Now *comment* out the `disp(filename)` line, and add a new line to load in each dataset from the new filename that is constructed each time the loop goes around and assign it to the `temp` array. 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.) Look at the **Workspace window** – note that you have an array (`temp`) that has size 94×192×12. If temp is 94×192×1 then go back a page or so and go through the bit about loading data into a 3D array. You want to avoid over-writing the information that is already there, so the line; `temp = load(filename);` will not work (and you will only get a 94×92 array after going 12 times around the loop). Why? (Again, look back a page-ish.)[16]

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

Now ... before the loop, create the (lon,lat) information arrays using the `meshgrid` function. Add the necessary line(s) to the script (after the workspace initialization but before the loop starts) to create the (lon,lat) information arrays. Note that you only need 2D arrays here because the same (lon,lat) can be used to plot the temperature data for each month.

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

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

Don't just type this line in blindly (maybe it doesn't 'work' anyway). Make sure that you understand what you are doing (otherwise why do GEO111 at all?).

Save and run the script. Do you have 12 different temperature plots on the computer screen?[17] Note that this is where the close all command at the start of your script comes in useful. Because if you re-run the script, you wont then end up with 24 figure windows. And then 36 the time after that, and ... (There is actually no need to create a new figure window each time – comment out the command that creates a new figure window (figure). Save and re-run and note the difference.) Get the units of the temperature data array into units of °C or °F rather than °K. Either: assign the `temp` array data to a new array and make the appropriate conversion from °K (all within the loop), or you can do this subtraction on the line that you actually plot the data (i.e., within the `contour/contourf` function), for example:

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

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

would convert to °C as it plotted the data.

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

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

and then to the `contour(...)` (or `contourf(...)`) function, add `,v` to the end of the list of passed parameters. This particular choice for the vector `v` tells MATLAB to do the contouring from -40 to 40 (°C), and at a contour interval of 2 (°C).. Play around with the min and max limits of the range, and also with the contour interval to see what gives the clearest and least cluttered plot. For instance, maybe you don't want the low temperatures to go 'off' the scale (the white color in the filled contour plot).

Finally, finally ... look up **MATLAB** `help` on `getframe`. Then go back to your global temperature loading/plotting script and add the following line[18]:

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

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

```
» movie(M,5,2)
```

and hopefully ... an animation of the progression of monthly surface air temperatures globally, should appear[19].

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.

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

NOW FOR A MORE COMPLICATED EXAMPLE but still using the global temperature dataset. It would be nice, to add to the temperature distribution plot, the continental outline. Currently you are left to some extent guessing where the land and where the ocean is, although the temperature contours to delineate the boundaries remarkably well in some places (depending on how many and hence density of contours has been specified).

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

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

Load these into the **MATLAB** workspace (in the 'usual way'). You should now have 2 vectors. Maybe view then in the **Variable Window** to get a better idea of what you are dealing with. Also keep an eye on the entries in the **Workspace Window** and particular the **Min** and **Max** values – looking out for any indication of `NaN`s or negative

`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.

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

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

*** *the* `for ... end` *loop* ***

numbers.[20] Try plotting these lon/lat locations. Use the `scatter` plotting function (which makes it all the easier as your data is in the form of 2 vectors already). You might need to reduce the size of the plotted points (refer to the earlier exercises, or `help`) and additionally, you might want to fill the points (up to you). Remember you can set the axis limits, which presumably should be 0 to 360 or -180 to 180, on the x-axis (longitude), and -90 to +90 on the y-axis (latitude). Font sizes of labels can also be increased if necessary. You might end up with something like Figure 3.2.

By plotting a 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 3.3..

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 points, which is why you get the straight lines criss-crossing the map and joining each successive continent and island in the dataset.

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
```

These vectors hold information regarding the start row and end row, or each shape. Again, view the contents of these vectors to get an idea of what is going on. For example, you'll see that the first entry is that the first shape starts on row `1` (`continental_outline_start.dat`), and ends on row `100` (`continental_outline_end.dat`). The 2nd shape starts on row `101`, and ends on row `200`. etc etc The simplest way too start dealing with all this, is to just plot the very first shape, defined by rows `1-100` of the lon and lat vectors. By now, you hopefully will be able to see that to plot rows `1-100` of lon and lat data, you are going to do:

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

(here I have named the arrays `lon` and `lat` for added convenience).

Well ... this is probably about as unexciting as it gets – a small piece of the Antarctic coastline. If you did `hold on` and plotted the next block (rows `101-200`), you get the next chunk of coastline. (Try this and see.) You could keep going this – manually adding additional sections of the global continental outline. This could get tedious ... and it turns out that there are 283 different fragments to plot, all one after another. (This number comes from asking **MATLAB** the `length` of `continental_outline_start.dat` or

[20] QUESTION: How else (from the command line) would you determine whether there were any `NaN`s in the vector? Equivalently, how would you determine whether there were any values less than zero? Try both.

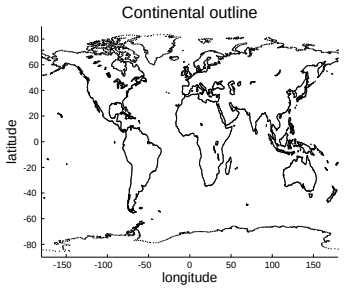


Figure 3.2: Continental outline (of sorts).

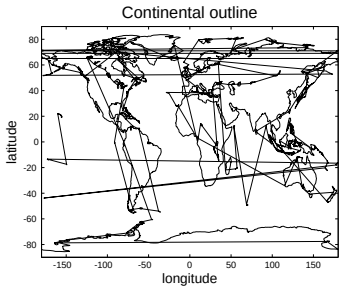


Figure 3.3: Another continental outline (of sorts).

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

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

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

where for the first fragment, the values of `line_start` and `line_end` are given by `lstart(1)` and `lend(1)`, respectively (renaming the original vectors to shorten the variable name)[21]. 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 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[22])?

So ... create yourself an `m-file`. Load in the lon-lat pairs as vectors (renaming then to something more manageable if you wish). Load in the vectors continuing the start and end information. Create a `do ... end` loop. Maybe print (`disp`) the loop count and run the program (after saving), just to check first that the loop is functioning correctly. Before the loop, create a Figure window. Set `hold on`. In the loop all you need is the plot command, but with the start and end rows being a function of `n` (or whatever you call the loop counter). Set axis dimensions and label nicely (after the loop ends). Run it. Hopefully ... something like Figure 3.4 appears(?)

`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).

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

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

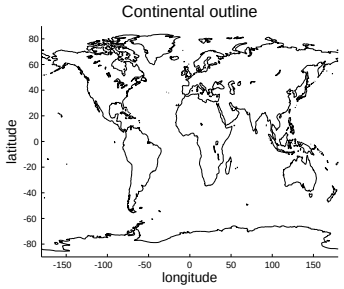


Figure 3.4: Another go at the continental outline!

3.3 *Sub-programs (scripts)*

*** *sub-programs* ***

An example involving the continental outline and now combining with the global temperature dataset/plot.

Go back to the script you wrote for creating the global temperature map animation. Copy it and rename it, and remove the loop and also the creation of the movie, so that it simply loads in a single month of data (any one), creates the lon-lat info (`meshgrid`), plots the

temperature field and makes the plot 'nice' (labels, and maybe an optimized number of contours and color scale), i.e. ending up when you run your script, with something looking like Figure 2.15. (Make sure you save this before moving on.)

Now lets say that you want to add the continental outline as an overlay. In fact, you do want to do this! You could certainly add to your temperature field plotting script:

1. Loading in of the lon-lat, and also start-end, vector data.
2. A `hold on` after the temperature data has been plotted.
3. A `do ... end` loop, to plot all the coastline fragments.

In fact, this, itself, is worth trying. Save it with a different filename and run it. You should end up with a nice outline of the continents/islands on top of the contoured map. (You could also try plotting the outline first before the contour function, particularly if using the filled function (`contourf`) – what happens?)

But lets imagine that you might be in the habit of plotting lots of different global datasets, and for each, you want the continental outline. You would have to put exactly the same code in each and every script you write. There is a better way of dealing with this situation (i.e. a block of code that you might want to use again and again as part of different programs and projects).

You can place a block of code that you want to re-use, in its own `m-file`. Go back to the script that you wrote to just plot the temeprature field (no continental outline overlay). Create a new (blank) m-file and place into it:

1. Loading in of the lon-lat, and also start-end, vector data.
2. A `do ... end` loop, to plot all the coastline fragments.

Save it.

At this point it is a good idea to test it rather than immediately trying to combine it with another complex script. Lets say that the code to load and plot the continental outline is called `plot_continents` (filename: `plot_continents.m`). Test it by opening a figure window, setting hold on, and then calling (running) the script, e.g.:

```
figure;
hold on;
plot_continents
```

and hopefully getting a version of Figure 3.4 but without the fancy labels etc. The next, trivial but oddly profound step, is to place the above 3 lines of code in a new `m-file`, and then run it. Now you have created a program that calls a sub-program (`plot_continents`)! (One might classify the 3-line program a test harness for the sub-program

– i.e. just enough commands to make the sub-program work and thereby have verified that all seems fine with it.)

Now the last and genuinely trivial step is to call `plot_continents` from your temperature field plotting program, either just after the contouring function and a hold on has been called, or if you prefer, after the plot has been labelled and the axes limits set.

A little refinement here is to increase the line weighting to e.g. a 1.5 pt width to make them a little more pronounced compared to the color contoured background, by adjusting the plotting line thickness (and also ensuring it is black):

```
plot(lon(lstart(n):lend(n)),lat(lstart(n):lend(n)), ...[23]
ḱ-','LineWidth',1.5);
```

The only continental-scale fly in the plotting ointment now, is illustrated in Figure 3.5. It may be a little hard to se, but the continental outline has only been plotted from 0 to 180E, despite the plotting subroutine having been checked (and dutifully plotted a global distribution) earlier. How is this possible?

If you compare the separate plots – Figure 2.15 vs. Figure 3.4, it is apparent that the first goes from 0 to 360E, and the latter from -180 to 180E. Hence when combined on a 0 to 360E scale plot, the -180 to 0E portion of the continental outline has been lost. A crude fix for this is to plot the continental outline *twice*, with one version offset by 360 in longitude, i.e. you end up with two, side-by-side copies of the outline, spanning from -180E to 540E (= 360 + 180), which when restricted to 0 to 360 leaves you with a complete outline (the excess parts having been clipped by the `axes` command and not displayed anywhere). To do this, the key line in `plot_continents` is duplicated and 360 added to the longitude values in the 2nd version:

```
plot(lon(lstart(n):lend(n)),lat(lstart(n):lend(n)), ...
ḱ-','LineWidth',1.5);
plot(lon(lstart(n):lend(n))+360,lat(lstart(n):lend(n)), ...
ḱ-','LineWidth',1.5);
```

And now it is fixed. Time for beer.

3.4 *Conditional statements*

One of the other main programming constructs is the conditional statement, in which the outcome (one or more statement(s)) is conditional on the 'truth' or otherwise of a given (i.e. it being true or false). This is embodied in **MATLAB** (and similarly in most languages) by the `if ... end` construct (see **Conditional Statements** Box).

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

[23] Note the `...` notation (see Box).

...

Three points in a row in MATLAB – `...` — at the end of a line, indicates that the next line should be treated as a continuation of the current line. i.e. it is a way of breaking an overly long line into two fragments without **MATLAB** thinking they are two completely sperate and independent lines. e.g. trivially one could write:

```
hold ...
on
```

Pointless. But valid.

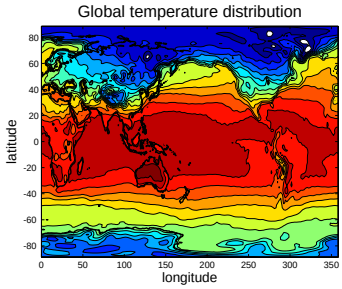


Figure 3.5: Now continents on top of temperature fields.

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
```

(See `help` for a further variant including `elseif`.)

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.

A rather trivial example would be to modify the earlier movie generating program, to omit the month of July. July is the 7th month in the sequence, and hence value of the *loop* counter when the July data is loaded and plotted. As an exercise ... add an `if ... end` conditional statement construct to your code, such that only when `n` is not equal to `7`, is a file loaded and the data plotted (and added to the movie).[24]

*** *the* `if ... end` *conditional statement, test for inequality (~=)* ***

[24] HINT: In the `if`, you need to test for the month number counter, not being equal to 7 (July).

A more practical example would be to test for a filename already existing and if so, automatically modifying the new file name so as not to over-write the file.[25] The relevant function is `exist` and in the case of a test for a file, returns either `0` (the file does not exist in the MATLAB search path, although that does not rule out it existing somewhere else entirely), or `2` (the file exists). Clearly(?), before you save the movie file, you 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. If not, and the filename has not already been used, you can proceed as before – the equivalent of 'doing nothing'.[26] Go ahead – try it (i.e. modify your code to avoid over-writing an existing filename).

*** *more on the* `if ... end` *conditional statement, test for equality (==), introducing the* `exist` *function* ***

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

`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.

3.5 *Functions*

Creating *functions* in **MATLAB** are very much like the subprogram script from the previous section ... except a *function* returns one or

more values (see Box). *Functions* are basically what you have already been using the entire time whether or not you realized it. For example, plot and scatter are in fact a functions, and return the ID of the plot graphic. We simply have not been asking for the returned value so far. As per **MATLAB** `help`:

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

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

As an example, – go re-load the bathymetry data one last time (unless you already have it in your variable workspace). You have already plotted just land on its own in a previous Example. How many land cells are there? You'll need to use the `find` function, and also then obtain the number of locations meeting this criteria[27].

But what about: what fraction of the Earth's surface is land? What (area) fraction of land is within 70 m of the current sealevel? For these questions, simply counting cell above and below sealevel, or within a certain band of height, is not enough. Why? Because the area of each cell shrinks towards the poles as the distance around the Earth along a line of longitude becomes progressively less. For any particular cell in your dataset, with Westerly and Easterly longitudinal limits of lon_W and lon_E, respectively, and Southerly and Northerly latitude limits of lat_S and lat_N, respectively, its area is:

$$2 \cdot \pi \cdot R^2 \cdot [sin(lat_N) - sin(lat_S)] \cdot (lon_E - lon_W)/360$$

where R is the radius of the Earth – assume 6,371 km, or 6.371×10^6 m.

The first thing to note here is that MATLAB does it calculations of trig functions such as `sin` and `cos`, with the input in units of radians, not degrees. So if you are working in degrees, convert to radians by dividing by 180 and multiplying by `pi`[28].

If you now, using `meshgrid`, create the longitude and latitude matrices to go with the bathymetry data matrix, you can write out the formula and pass in any pair of bounding longitudes and latitudes for a cell. Remember that the cell centers go:

```
0.5, 1.5, 2.5 ...  359.5
```

for longitude, and

```
-89.5, -88.5, -87.5 ...  89.5
```

for latitude, while you want the edges, which will be $\pm 0.5°$from the centers in both longitude and latitude.

functions

The script file for a function in **MATLAB** has a special header line at the very top of the `m-file`. As the **MATLAB** on-line documentation says:

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

"declares a function named `myfun` that accepts inputs `x1,...,xM` and returns outputs `y1,...,yN`. This declaration statement must be the first executable line of the function" (which I already said!).

This general form of description is not entirely un-contorted. So for instance, trivially, a the `m-file` of a *function* to calculate the square of a number and return the value, would look like:

```
function [y] = ...
mystupidfunction(x)
y=x^2;
```

and called by e.g.:

```
» mystupidfunction(2)
ans =
4
```

Functions are named and saved with the `.m` extension just as per normal *script* files.

*** *nested loops, functions, `meshgrid`* ***

[27] HINT: The length of the returned vector.

[28] Remembering that `pi` is a built-in constant with a value of 3.141592653589793 ...

Taking the example of total land surface area – how much is there, in m^2? You could actually write this in just 2 lines of code in MATLAB if you are clever ... but her we are going to do this via a nested loop. To start with, you are going to create a nested loop (i.e. one loop inside the other):

```
for n=1:360
   for m=1:180
   end
end
```

What this is doing is looping through all 360 columns (i.e. longitude), and for each column, looping through all 180 rows (i.e. latitude), with the effect that every grid point is passed through. You could check on what this is going by adding something like:

```
   disp(['(n,m) = ' num2str(n) ',' num2str(m)])
```

in the inner loop (which simply creates and then displays a string, telling you the n and m value pair).[29]

From this, you can derive the grid point centers (in degrees) by:

```
lon = n - 0.5
lat = m - 0.5 - 90.0
```

This should be obvious ... ? Note that latitude starts at -90°N, hence the need for the `-90.0` subtraction. Your cell boundaries are then:

```
lon_E = lon + 0.5
lon_W = lon - 0.5
lat_N = lat + 0.5
lat_S = lat - 0.5
```

and hence the area:

```
2.0 * pi * (6.371E6)^2 * ...
(sin(pi*lat_N/180.0) - sin(pi*lat_S/180.0)) * ...
(lon_E - lon_W)/360.0
```

This is a sort of useful calculation and there may be a variety of instances where you require knowledge of the area of a cell on the Earth's surface. So we are going to put this fragment of code into a *function*, passing the Westerly and Easterly longitudinal limit, and Southerly and Northerly latitude limits and returning the calculated area. For header for the function will look like:

```
function [area] = calc_area(lon_E, lon_W, lat_N, lat_S)
```

The line calculating the area is the only essential content of this m-file (`calc_area.m`). Note that the result of the calculation must be assigned to a variable `area`, matching the function header (or you will get nothing returned for your trouble), i.e.

[29] Remember that you could test this nested loop with a smaller range for `n` and `m` first, e.g. `1:36` and `1:18` (or even smaller).

```
area = 2.0 * pi * (6.371E6)^2 * ...
(sin(pi*lat_N/180.0) - sin(pi*lat_S/180.0)) * ...
(lon_E - lon_W)/360.0
```

Also note that although this is the minimum content, it is good practice to adequately comment the code. Also, as a further refinement, you might define a variable to hold the value of the Earth's radius so that the equation becomes easier to interpret, i.e.

```
earth_radius = 6.371E6;
area = 2.0 * pi * earth_radius^2 * ...
(sin(pi*lat_N/180.0) - sin(pi*lat_S/180.0)) * ...
(lon_E - lon_W)/360.0
```

Other possible alternatives include passing just the (cell centre) lon and lat values, and deriving the cell boundaries from these. This would necessitate assuming that the grid is 1° in both directions (and hence make the function less generic and applicable to other problems). Or one could pass in the lon, lat pair, plus a 3rd parameter for the resolution (here, 1.0°). There are lots of alternative possibilities, all 'correct' for this specific example, more or less complicated and with more or fewer input parameters, and more or less applicable for other situations.

So now we are close to determining the total land surface area. The *conditional* expression, within the loop, should be obvious ... [30]? All that then remains is to sum up the area of each cell that meets the criteria (of being above sealevel). Again, there are various ways to accomplish this:

[30] HINT: You are testing for the topographic height being `> 0.0`, or perhaps `>= 0.0`.

1. You could populate a 2D array, the same size as the bathymetry data (360×180), and set each cell to its respective area, if the cell is above sealevel, and to zero if below. If this array was called: `land_area`, then the total global area of land would be:

   ```
   sum(sum(land_area))
   ```
   [31]

2. Unless you will need the individual areas again, it is not necessary to save them all explicitly. Instead, we could generate a running total by adding the cell area to a variable each time a land cell is found. If the running total variable was called `area_sum`, then at each identification of a land cell, in the `if ... end` structure, we would write:

   ```
   area_sum = area_sum + calc_area(lon_E, lon_W, lat_N, lat_S);
   ```

   What this is saying is: take the existing value of `area_sum`, and add the value calculated by `calc_area` to it.
   It is not necessary here (in **MATLAB**), but it is good practice to initialize this variable – somewhere before the nested loop, you would do this by writing:

[31] If it is not obvious that this is the case, check on the details of `sum` in `help` (and what it returns if passed a matrix).

```
area_sum = 0.0;
```

Some programming languages (e.g. **FORTRAN**) are fussy about variables being explicitly initialized with something to start with.

Also try modifying your script to the *fraction* of the total land area potentially threatened by future sea-level rise (assume: 70 m). You could do this by creating and updating 2 partial sums – one for land are below 70 m, and one for total land area (as before). Simply divide one by the other (and maybe multiply by 100 to get a % area fraction).

3.6 *Algorithms and problem-solving*

*** *loops, algorithms* ***

In the 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° grid cell). A more involved question might be: how many distinct land masses are there?

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 (modern) topography of a simple Earth system model ('GENIE'). Actually, you are only going to be concerned with the land-sea mask and not even worry about height above, or below, sea-level.

Load in the file: **genietopo.dat** 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[32], and values above 90, land[33] – 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 3.6.

You are going to count up (and sequentially number) the different land masses[34]. 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.

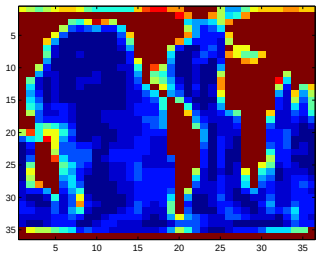


Figure 3.6: Ocean topography (blues through red) in the 'GENIE' Earth system model. Land is shown in brown.

[32] 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.

[33] The values: `91`, `92`, `93`, `94`, represent different compass directions of runoff on land. (another not interesting and barely useful fact.)

[34] 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.

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(genietopo);
```

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('genietopo.dat','-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

size

`size` returns the size of an array, as a vector of length n, where n is the number of dimensions of the array.

For a matrix, a 2-element vector is returned with the values corresponding to the number of rows and the number of columns (in that order). These values can be handily saved by assigning the result of size to a paid of (scalar) variables:

```
» [n_rows n_cols] = ...
  size(MATRIX)
```

where `MATRIX` is the matrix array name, and `[n_ronws n_cols]` forms a 2-element vector to be assigned the result to.

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.

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('genietopo.dat','-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
```

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. [35] (It is quite pretty to plot though, as per Figure 3.7. 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[36].

[35] 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.

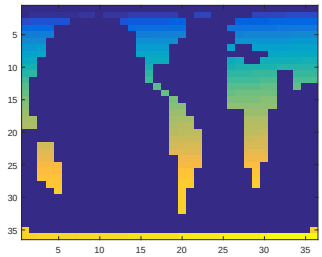


Figure 3.7: The 'GENIE' mode land grid, with land points assigned a sequential integer (working across and dow the grid – from West to East, and then North to South).

[36] 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.
```

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. 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 flowchart 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[37]:

```
if ( (land_id(lat-1,lon)==0) && (land_id(lat),lon-1)==0) )
end
```

[37] Not all of these parentheses are necessary – I have written it like this to make the conditional (hopefully!) completely clear.

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[38]. 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 of structuring this, some using more and some less, code. One possibility (and not necessarily the most optimal one) is:

[38] **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.)

```
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[39]. 'CODE BLOCK #4' will use the conditional code that you saw earlier:

[39] 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.

```
if ( (land_id(lat-1,lon)==0) && (land_id(lat),lon-1)==0) )
end
```

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 is 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
```

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
```

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`).

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 3.8. Soooooo close[40]. 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 3.9) and similar (but different – why?) artifacts arise (plus now one cell in Antarctica has a different color from the rest of the continent).

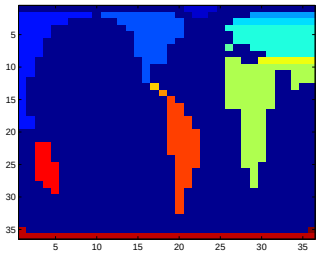


Figure 3.8: The 'GENIE' mode land grid, with land points assigned a unique identifier ... almost ... (!)

[40] 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.

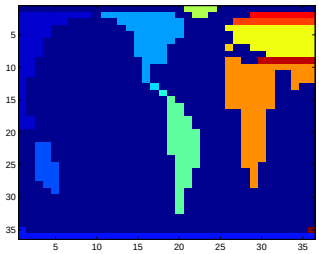


Figure 3.9: The 'GENIE' mode land grid, with land points assigned a unique identifier (color).

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 `4`s 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 3.8 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 (`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 3.10.

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:

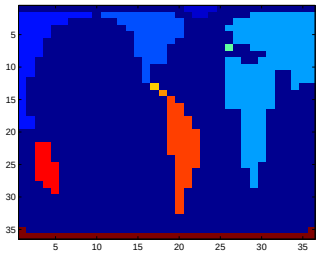


Figure 3.10: The 'GENIE' mode land grid, with land points (almost) assigned a unique identifier (color).

```
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 3.11 and now represents a complete solution to the problem.

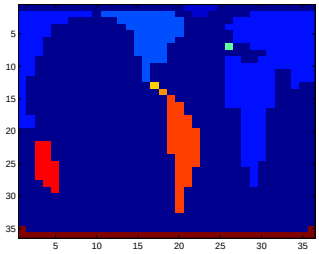


Figure 3.11: The 'GENIE' mode land grid, with land points assigned a unique identifier (color).

[41] Because we re-numbered them earlier, right?

[42] 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.

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[41]. 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`. [42]

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.

4
Introduction to numerical modelling

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.

This Lab's porpoise is to get some familiarity (== play) with computer models. You will see what time-stepping is in numerical modelling, and where some loops might just come in handy.

4.1 Introduction

As an example, we will consider a simple population model. 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).

First, consider the simple model:

$$N_{(t+1)} = \lambda \cdot N_{(t)}$$

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.

- $N_{(t)}$ ... is the size of the population at time t.
- $N_{(t+1)}$ ... is the size of the population at time $(t+1)$.
- λ ... is the average number of offspring produced, per adult per year, less mortality.

Don't get put off by all the Ns and subscripts and things. All Equation 1 says is that the population size (number of individuals = N) at some time in the future (time = $t+1$) is equal to the population now (time = t) multiplied by some factor. This factor is given the

Greek letter λ. If the units of λ were per year (yr^{-1}), t would then represent the time in years since the start (of the model).[1] The factor λ 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.

[1] We could equally write this in terms of the generation number.

So, we are simply asking; how many individuals will there be next year (time = *t+1*)? The answer is; the same number as currently (this year, or time t), minus the fraction of the population who snowboard off of a cliff or die of old age, $\alpha N_{(t)}$, plus the number of births in the population, which is also assumed proportional to the current number of individuals in the population, $\beta N_{(t)}$ (we are ignoring the time between birth and being ready to produce offspring in this equation).

If there are $N_{(t)}$ individuals at present (time = t), the number next year (t = *t+1*) is:

$$N_{(t+1)} = N_{(t)} + \beta \cdot N_{(t)} - \alpha \cdot N_{(t)}$$

Re-arranging this we get:

$$N_{(t+1)} = (1 + \beta - \alpha) \cdot N_{(t)}$$

The only even faintly subversive thing that has happened to te 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 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 time-step away. You estimate the change in quantity (rabbits, snowboarders, cloud water droplets) that occurs over the course of one time-step, and add it to the current quantity. Because you then know the quantity at time *t+1*, you can go calculate the quantity at time *t+2*. Then, knowing the quantity at time *t+2*, you can calculate the quantity at time *t+3*. Then, etc etc etc. Zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz (Is this sounding like a mission for loops, yet?)

This model predicts that as long as $\lambda > 1$, the population will increase exponentially, year by year, 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 λ 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, 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.[2] 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.

[2] This of course rather depends on the solution converging and not oscillating or exponentially growing ...

You'll probably want to plot the results. [3]. And you may want to save the data of population number vs. population (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.

[3] 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

---

In a variant of this example ... one might consider that most animal 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:

- 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 Equation 1, to give:

$$N_{(t+1)} = \frac{\lambda \cdot N_{(t)}}{(1+a \cdot N_t)^b}$$

There are two new parameters here:

1. b ... defines the strength of the density dependence and the dynamics of the population, and
2. a ... is a scaling factor.

Try starting with values of:

save

If you just type `save` and specify a filename, the entire **MATLAB** variable workspace is saved as a **.mat** file called **FILENAME.mat**.

More useful is the form:

```
save FILENAME VARIABLE ...
-ASCII
```

which save the variable `VARIABLE` in the file `FILENAME`, as plain text (ASCII).

- $\lambda = 2.0$
- $b = 0.1$
- $a = 0.1$

and run for e.g. 100 or 1000 years (/generations). Then systematically investigate the effect of changing the value of parameter b on the dynamics of the population, keeping the values of the parameters λ and a constant.[4] 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:

[4] This sort of exercise is know as a sensitivity analysis – i.e. quantifying the sensitivity of the model behavior or final result, to the value of a particular parameter.

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).

Don't spend too much time playing. I know how much fun you are having ;)

This is a genuine 24-carat time-dependent (time-stepping) numerical model, although it doesn't seem that exciting. You can see that the population value at each subsequent time ($t+1$) depends directly on the value at the previous time (t). Could you predict the population size far into the future (large t) analytically (i.e., write down an equation and solve it)? Note the use of parameters, whose values can be easily updated or passed directly into the script and instantly affecting the entire model as well as updating the graphical display. Pretty useful eh?

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!

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.

4.2 *Box models*

*** *matrix maths* ***

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 Americas 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 4.1, together with the mean volumes of the lakes and the annual flow rate of water out of them.

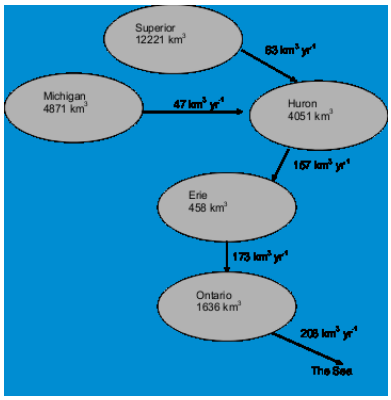


Figure 4.1: Lake volumes and river flow rates in the Great Lakes system.

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.

Table 4.1: Pollution input rates to each of the 5 lakes.

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

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 (kg km^{-3}) of heavy metals in the lakes; `cS`, `cM`, `cH`, `cE`, and `cO` (for; Superior, Michigan, Huron, Erie, and Ontarion, 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 1.0×10^3 kg yr^{-1} (1000 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 km^3 yr^{-1}.

The metal outflow flux is then just the concentration of metals in the water (`cS`), times by the water flow; `63*cS`. Thus, for Lake Superior, we can write `1000 = 63*cS`. The other lakes can be similarly analysed, to give a set of 5 equations:

```
1000 = 63*cS
4500 = 47*cM
1000 + 63*cS + 47*cM = 157*cH
3500 + 157*cH = 173*cE
3000 + 173*cE = 208*cO
```

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:

```
 63*cS +   0*cM +   0*cH +   0*cE +   0*cO = 1000
  0*cS +  47*cM +   0*cH +   0*cE +   0*cO = 4500
-63*cS + -47*cM + 157*cH +   0*cE +   0*cO = 1000
  0*cS +   0*cM - 157*cH + 173*cE +   0*cO = 3500
  0*cS +   0*cM +   0*cH - 173*cE + 208*cO = 3000
```

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 5×5 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 5×1 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:

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

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 5×1 vector C). And **MATLAB** will give you the inverse of R (if it exists) on a plate.[5] 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.[6]

[5] 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.

[6] Note that the equations above are written in normal maths language, e.g. with a $\times$ rather than the $*$ that **MATLAB** understands.

We can tackle this example in a different way. Instead of looking at the steady-state solution of the system, you might want to follow how the concentrations of heavy metals in the 5 lakes changes with time, perhaps as a result of a change in river flow, or a pollution incident. There is no analytical solution to the time-dependent response of the Great Lake system to a perturbation, so you are going to have to time-step through the simulated lake system, calculating the net change (i.e., loss or gain) of heavy metals for each of the lakes at each time-step. This will also illustrate something about what constitutes an equilibrium situation (or steady state) in a computer model – you never ever *quite* get there, so some judgement is required as to where you draw the line and go off down the bar.

To create the time-stepping model of the Great Lakes system:

1. Create a new **MATLAB** script.
2. Assume initially that each time step represents 1 year. Start with a relatively short loop so that you can sensibly print stuff out to the screen (`disp`) to follow whether things are working and see what is going on. If you call the loop variable `t`, then a loop with `t = 1:10` will be sufficient to start with. Remember: start simple so that you can follow what is going on and debug it, and only later try and extend the (working) script to its final level of complexity.

*** *loops* ***

3. Before the start of the loop (but after you have commented what it is all going to do – you weren't going to forget this were you ... ?) create 5 sperate variables (or a vector of length 5 if you are feeling brave) to store the initial concentrations of heavy metals in the lakes and assign them all a value of zero.[7] You might end up with something like:

```
cS = 0.0;
cM = 0.0;
...
```

4. In case we ever need to change the input fluxes of metals to the lakes, we will use parameters to store the values of these fluxes. The values are given in Table 4.1. Define the parameters (again, before the start of the loop) something like:

```
fS = 1000;
fM = 4500;
...
```

(or as a vector) and don't forget to include a comment regarding what the units of the fluxes are ...

5. Similarly, define the 5 parameters containing the river outflow rate from each lake:

```
rS = 63;
rM = 47;
...
```

6. You need one final piece of model initialization and also define the volumes of the lakes as parameters. Although the lake volumes might never change much, it is generally neater in computer code to have parameter names in the equations rather than numbers. And if you ever need to use the value again, it is easier to remember the parameter name, then have to go look up the value to write down each time (particularly if it is a number with lots and lots of digits). The volumes of the lakes are given in Figure 4.1, so something like:

```
vS = 12221;
vM = 4871;
...
```

You are now ready to start coding the main part of the numerical model. The first thing to do is to add the heavy metal inputs to each of the lakes. Because the metal inputs are fluxes (i.e., the input is continuous), this addition must be done at each and every time step. (How initial non-zero concentrations gradually decline over time, even with no further input, is equally a task for numerical simulation.) You should see that the code for making this metal input to the lakes will need to go inside of the loop structure. Yes? (Each successive year and each new time around the loop you will want to add a

[7] This is called initializing the variables. In some programming languages (such as FORTRAN) it can be very important to formally set variables you have created to zero before you start using them.

new bucket-load of metals pollution into the lakes.) OK, then we will proceed:

→ We know the concentration of metals in each of the lakes from the previous time-step *t*. We want to update the concentration for time-step *t+1*.
→ The new (time *t+1*) concentration of metals in the lakes will be equal to the previous concentration (time *t*) PLUS the change in concentration (Δconcentration) that reflects the addition of heavy metals from the surrounding urban areas (we'll worry about the losses later). Δconcentration is equal to the amount you put in (Δmass) divided by the volume of the lake (it is as simple as; Δconcentration = Δmass / volume). Although Δmass is actually flux × Δtime, because the time step (Δtime) has length of 1 year, for now you can write; Δmass = flux × 1, or Δmass = flux.
→ For each lake, take the concentration variable, add the increase in concentration (Δconcentration) to this variable, and assign this new value back to the concentration variable. The code for Lake Superior should look something like:

```
cS = cS + fS/vS;
```

→ At this point it would be sensible to check on how it is working so far before adding add any more code. [8] [9] The result of your efforts so far, should be a series of lake concentrations that simply increase year on year.

Your next step could be either: (a) complete the model, or (b) add some graphical display. Arguably, adding the display first is more useful as it aids in debugging the model. But to plot how the metal concentrations evolve with tim, you will need to first store the information needed for the plot in an array. If the array is called `hello_kitty` (but please call it something more useful than this ...) then at the end of the loop you would write:

```
hello_kitty(t,1) = t;
hello_kitty(t,2) = cS;
...
```

All this is saying is that in the first position (column #1) of row t we assign the time-step number (which is equal to the year since the start of the model). In the second position (column #2) we will assign the concentration value, etc.[10]

Go run this. You should notice that an array `hello_kitty` appears in the **MATLAB Workspace** window. It should have size (**Value**) of 10×6. Look at the array contents in the **MATLAB Array Editor** (or type `size(hello_kitty)`) and check that the concentration values

[8] For printing our debugging info to the screen, you could do this in bits, e.g.:

```
disp('year') disp(t)
```

or neater would be to convert the number value to a string, and then concatenating it with some sort of informative text label:

```
disp(['year = ' num2str(t)])
```

Equally you could print out concentrations:

```
disp(['concentration = ' ...
num2str(cS)])
```

[9] Equally, you could add a breakpoint within the loop, allowing you to step through the loop one iteration at a time.

[10] Make sure that you understand why the `t` appears in `hello_kitty(t,1)` (i.e., how we are using the increasing value of the loop counter variable t to keep adding new rows to store the newly updated concentration values).

are the same as you were previously printing out to the screen. Go debug if it if things are screwed up :(

At the end of the script plot the graph of time (year) against concentration. We could plot all the data on the same graph to compare what is going on. Maybe something like:[11]:

```
hold on
plot(hello_kitty(:,1),hello_kitty(:,2),'k-')
plot(hello_kitty(:,1),hello_kitty(:,3),'r-')
plot(hello_kitty(:,1),hello_kitty(:,4),'g-')
plot(hello_kitty(:,1),hello_kitty(:,5),'b-')
plot(hello_kitty(:,1),hello_kitty(:,6),'c-')
```

[11] The code to get a continuous black line for Lake Superior can be passed as a parameter to the plot function – it is `k-` (see **MATLAB** help on **LineSpec**). The 2nd, 3rd, 4th, and 5th lines are colour-coded – red: Michigan; green: Huron; blue: Eries; and cyan: Ontario.

Make sure that everything is labelled etc as it should be. There are no prizes in life for being a lazy muppet.

Be honest – is this not the most boring graph you have ever seen? Increase the number of time-steps the model runs through to 1000 (i.e., in the for loop definition). Re-run. OK; so it hasn't got any more interesting ...

What is missing how pollutants leave the lakes. This is why the concentrations just rise linearly for ever and ever and ever. Losses to the lakes occur because the lakes drain out, either into other lakes, or (in the case of Lake Ontario) to the sea. We need to represent the losses of heavy metals from each of the lakes due to this drainage. The flux of metals out of each lake will be equal to the concentration of metals in the water times the water flow rate. Yes? So for Lake Superior, the loss of metals from each lake due to drainage will be equal to `rS*cS`.

The units of loss are kg yr^{-1} just like the flux input (units are kg km^{-3} × km^3 yr^{-1}, which equates to kg yr^{-1}). We can therefore simply update our equation to take into account drainage loss as well as flux input, e.g.[12]:

```
cS = cS + fS/vS - rS*cS/vS;
```

[12] If you wanted to, you could neaten up this equation by moving a factor of (`/vS`) outside of the net flux term (`fS - rS*cS`).

This Example does go on and on and on ... but very very almost there. There is just one final thing missing. When water flows into the lakes Huron, Erie, and Ontario, it is carrying with it heavy metal pollution from the lakes upstream. We need this last component in the model. For these three lakes we then need to introduce an additional flux. Fortunately, we have already calculated this flux – it is equal to the loss rate from the lake upstream. For example, the additional metal flux via river flow to Lake Ontario is equal to the river loss flux out of Lake Erie. Look at the diagram of the lake system to see how the lakes are connected. For Lake Ontario, for instance, we need to make one final modification to the mass balance calculation in the loop[13]:

[13] Again, you could shorten the equation if you want to by taking out common factors.

```
c0 = c0 + f0/v0 - r0*c0/v0 + rE*cE/v0;
```

All this say is that; the concentration of metals in Lake Ontario next year, will be equal to the concentration this year (`c0`) PLUS the metal flux input (`f0/v0`) MINUS the drainage loss (`r0*c0/v0`) PLUS the river input from lake Erie (`rE*cE/v0`). Also write the code to complete the mass balance for Lake Erie (gain from Lake Huron) and to Lake Huron (gains from Lake Michigan AND Lake Superior).

Re-run the script. Note that things are now much more interesting (ha!), and after 1000 time-steps (1000 years) the concentration of metals in the lakes levels off, although different lakes take different amounts of time level off (which takes the longest, and why?) and hopefully, something like Figure 4.2.

Try increasing the number of time-steps to 10000. Now you can clearly see that the concentration of heavy metals reaching what is know as steady-state (or it is in equilibrium), where the flux of metals in is exactly balanced by the flux of metals out and there is no longer any net change in the quantity (or concentration) of metals in the lake.[14] You should also note that as the system is currently set up there seems to be little point in running the model for more than 1000 or 2000 time-steps, because very little change occurs after that. It is up to you to decide that any further change that occurs is trivial and can be ignored – perhaps when 95% of the final change has been achieved, or 98% or 99%. It will depend on the situation and how computationally expensive the model is to run. If this model took 10 hours to run 10000 years, you would certainly say that 1000 years was enough. Whereas it is actually so quick that running out to 2000 or even 10000 years is not a problem. Obviously another consideration is whether readers of your pollution report would be at all interested in a graph that had the interesting change bunched up at the left-hand end and nothing happening for most of the plot. Just because you can calculate 1000000000s of years doesn't mean that you need present it to others. Some thought is required as to what information you want to get across in a presentation of a model simulation.

Now it is working (I hope) – adjust your script to make it into a function, and pass in a time-step. You will have to edit the equations because up until now, they have assumed by default, a time-step of one year. You will also have to remember that `t` stands for the time-step number, and no longer necessarily corresponds to time (the year). Fortunately, if you have followed the above instructions without trying to be clever(!) and do anything more efficiently, you already have a results array (apparently called `hello_kitty`) that explicitly stores the time at each time-step (and the value of time need not be the same as the time-step). Then explore what happens when you increase the time-step. For instance, you might see the

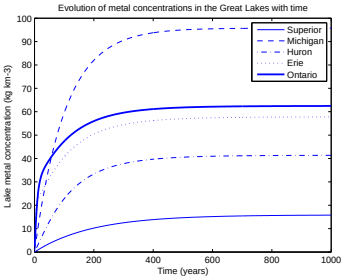


Figure 4.2: Simulated evolution of metal concentration in the Great Lakes system with time ... with labels that are far too small to make out :o)

[14] Check that the final steady-state (or close to steady state) metal concentration values are close to your analytical solution from before.

numerical solution start to get unstable as per in Figure 4.3.

As a further refinement, pass in a second parameter for the number of year the simulation should run for (and scale the plot to the same number of years). Remember that the total duration of the model experiment is equal to the time step duration times the total number of time-steps. Or in terms of setting the loop limit – the total number of time-steps is equal to the total duration divided by the time step duration.[15] You could also, instead of a maximum duration, pass in a criteria for the solution having approached steady state, and loop just enough times until this criteria is met.[16]

A final refinement would be to make better use of vectors, and rather than laboriously write out e.g.

```
fS = 1000.0;
fM = 4500.0;
fH = 1000.0;
fE = 3500.0;
fO = 3000.0;
```

you write:

```
f = [1000.0; 4500.0; 1000.0; 3500.0; 3000.0];
```

remembering that row #1 == 'S', row #2 == 'M', etc. You get much more compact code this way and it is much more scalable should there be additional lakes or reservoirs you might wish to include in a future version of the model.

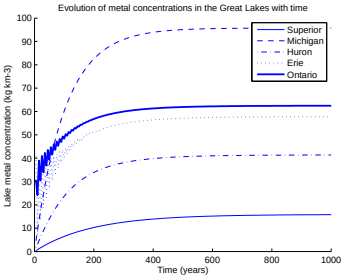


Figure 4.3: Simulated evolution of metal concentration in the Great Lakes system with time ... with labels that are far too small to make out ... and an integration time-step that is too long.

[15] Here we will ignore that fact that by dividing one number by another, particularly if both were reals rather than integers, we are going to end up with a real number, whereas the loop limit should really be an integer. (But **MATLAB** does not care and will work just fine.)

[16] A suitable criteria would e.g. be for the concentration in none of the lakes to change by more than x% per year.

4.3 *Energy-balance climate modelling*

Box, or zero-D models need not involve the reservoir of a substance (e.g. trace metal, carbon, or nutrient concentrations) *per se* – energy (heat) will do just fine. Which leads us to the climate system.

A RATHER LESS CONTRIVED EXAMPLE of a box model, is the global climate system, or rather, some measure of the (heat) energy in it.

To kick off – code up the analytical solution to the basic global mean energy budget at the surface of the Earth (see Box). (This will form the basis for subsequent more complex and time-stepping models.) You will need to find (Internet?) the values of the constants you need, and will need to be careful with units. Particularly temperature ... :o)

If you found a reasonable value for the solar constant, and did not screw-up the units on the Stefan-Boltzmann constant, then you should have an equilibrium (global, annual mean) surface temperature of around 14°C ...

Energy balance modelling (1)

The surface energy budget at the Earths surface, to a zero-th order approximation, can be thought of as a simple balance between incoming, sort-wave radiation that is *absorbed*, and out-going, infra-red radiation.

On average (over the Earths surface and annually), the energy flux received from the sun can be written:

$F_{in} = \frac{\alpha \cdot S_0}{4}$

(because the cross-sectional area of the Earth is $\frac{1}{4}$ of its total surface area).

Net outgoing infrared radiation proceeds according to black body emissions:

$F_{out} = \epsilon \cdot \sigma \cdot T^4$

where ϵ is the emissivity (ϵ=1.0) for a perfect black body radiator), σ is the Stefan-Boltzmann constant, and T the temperature in Kelvin. At least ... it would if there was no atmosphere and the Earth radiated directly from the surface to space. There is an (absorbing) atmosphere in the way! 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]:

$F_{out} = 0.62 \cdot \sigma \cdot T^4$

(with albedo: $\alpha = 0.3$).

Now turn your script into a function so that you can pass in the value of the solar constant and make it return the estimated global mean surface temperature for that value. You are going to make another script and call the function from this script. This new script is going to calculate the value of the solar constant S_0 (see Box), at 100 Myr intervals from 4.0 Gyr (4 billion years) in the past, to 4.0 Gry in the future – spanning approximately the age of the Earth and much of its potential long-term future. For the value of the solar constant at each time, you are going to call your function to calculate the corresponding surface temperature, and then plot mean global surface temperature vs. time. You'll need to:

- Step through time, from -4.0 to 4.0 Gyr, at 0.1 (100 Myr intervals).
- At each time, calculate:
  1. The value of the solar constant.
  2. The corresponding surface temperature of Earth.

  and store the calculated data as time (array column #1) vs. temperture (column #2).
- Plot your projected evolution of Earths surface temperature with time.

Likely areas of bugs include the units of time (Gyr), and that time is counted as the age of the Sun. Also be careful with nested parentheses (()).

Assuming that you have managed something like Figure 4.4 – 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'?

Solar constant

The long-term evolution of solar luminosity L_t as a function of time t can be approximated [*Gough* [1981]; *Feulner* [2012]) by:

$\frac{L_t}{L_0} = \frac{1}{1+\frac{2}{5}\cdot(1-\frac{t}{t_0})}$

where t_0 is the age of the sun – 4.57 Gyr (4.57×10^9 yr) and L_0 is the present-day solar luminosity (3.85×10^{26} W). This is equivalent to a flux (Wm^{-2}) of 1368 Wm^{-2} incident at the top of the atmosphere at Earth, which is given the symbol S_0. In the equation, L_0 can be substituted for S_0 to give the value of S_0 at ant time, i.e. S_t (Wm^{-2}).

Note that in the formula, t is counted (in Gyr) relative to the formation of the Sun (i.e. present-day would be: $t = 4.57$).

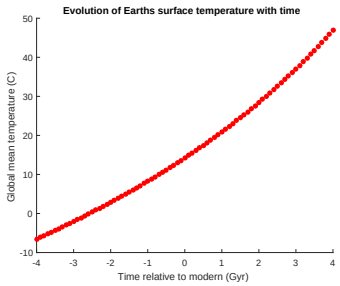


Figure 4.4: Simple EBM projection of the evolution of Earth surface temperature with time.

Now for daisies. Hell, why not?

So: there is an absolutely classic paper from the early 1980s – *Watson and Lovelock* [1983] – that illustrates how simple (biological) feedback on climate can lead to a close regulation of global climate over an appreciable span of the Earths past (and future). The premise for this model is a planet covered in bare soil (essentially, as per in the earlier EBM), but on which 2 different species of daisies (could be any pair of plants with contrasting properties) can grow – one white (high albedo) and one black (low albedo)[17]. 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

[17] As pointed out in *Watson and Lovelock* [1983], the actual 'colors' are immaterial – just tat the albedos differ.

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.

You are going to take your earlier, equilibrium EBM function, and develop it piece-by-piece until you have dynamic daisy populations interacting with climate and regulating the habitability of the planet. These are the steps (model code variants) you are going to work through:

1. Add a fixed fraction of two different sorts of daisy that modify mean global albedo and hence climate.
2. Add population dynamics to the two species of daisy, so they respond to changes in (global) temperature.
3. Modify the 'local' temperature of each species of daisy such that the growth of each differs for any given global mean temperature.

and for all, you are going to plot a pair of graphs to keep track of what is (or is not) going on.

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. We'll also start to worry about how much area is covered by each species of daisy.

Then make yourself some parameters for the model:

```
% define model parameters - daisy albedo
par_a_s = 0.3; % albedo - bare soil
par_a_w = 0.5; % albedo - white daisies
par_a_b = 0.1; % albedo - black daisies
% define model parameters - daisy land fraction
par_f_w = 0.01; % (land) fraction - white daisies
par_f_b = 0.01; % (land) fraction - black daisies
```

(using whatever parameter names you prefer). Here, the albedo values are fixed and will be used regardless of what the model does. The values have been chosen, assuming equal proportions of black and white daisies, to given an average of 0.3 – the albedo of bare soil and also the assumed value in the previous EBM. You'll modify and play with this value all too soon enough. The surface area fraction values are just initial values to start the model off.[18]

Next, you need to replace the fixed and assumed albedo value from before, with a variable whose value is calculated based on

[18] 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!).

`exit`

If you wish your program to end early, maybe because a parameter value check has revealed an inconsistency or an illegal value, or a calculation (typically an analytical solution) has failed or is impossible), simply add the command:

```
exit
```

(Note that in contrast, `quit` also exits the entire **MATLAB** program ...)

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 (bare soil then being 1.0 minus the combined daisy covered fraction). You weight the contribution to global albedo by the albedo of each daisy by its fractional area. You just then need to account for the fraction of the Earths surface that is bare soil (weighting the specific albedo of soil in the sum).

Re-run the model with the value of albedo now depending on the fraction of white and black daisies – it should look identically to before (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). You might play briefly with the prescribed daisy fractions and albedo values and e.g. check that when you specify a configuration with 100% of land area covered by black daisies, the climate is much warmer throughout the simulation, and when white daisies are assigned an initial value of 1.0, the climate is always much cooler compared to in the default simulation.[19]

Nothing exciting happening yet ... as per Figure 4.5 – the daisies stay at their prescribed fractional area and there is little, if any, impact on the gloal planetary temeprature[20].

OK – step #2, and now for the next modification and one which will actually make something 'happen' (i.e. the simulation will be different to that of the default EBM based simulation of mean global temperature response to increasing S_0. The daisies are going to grow and die, with their population changing over time until an equilibrium is reached (for a particular specified value of S_0). *Watson and Lovelock* [1983] give a simple population model formulation for the change in area fraction covered by both sorts of daisy with time (also see Box) that we will implement here.

In implementing these equations, note that the unit of population in Dairy World, is fractional area covered. So each time-step, the fractional area or each species will grow or shrink, depending on whether mortality is higher than growth. Both growth and mortality are formulated as being dependent on the fractional area (at the previous time-step), i.e. growth in covered area depends on how much is already covered. Similarly, mortality also depends on how many daisies are currently there. The growth rate is further modified by the available fractional area, such as that the area left shrinks, the growth rate shrinks. (Effectively, this is perhaps trying to account perhaps for shrinking resources available for further growth. It also has the effect of adding numerical stability to the model and helps presents over-shoots where the total fractional area covered by daisies far exceeds 1.0 ...).

If you have set this daisy population dynamics enabled EBM (a

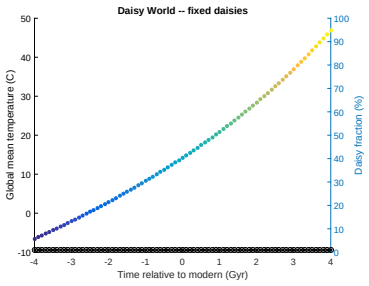


Figure 4.5: Evolution of global surface temperature and the two populations of daisies with time ... but with no change allowed in the daisy populations (d'uh!). The fractional coverage of white daisies is shown by large empty circles, and for black, by small filled black circles. Data points for mean surface temperature are color-coded by temperature (color scale not shown).

[19] Note that it is very easy to accidently prescribe a total area covered by daisies of >100%. You should ideally put a check (`if ... end`) in the code before it tries to calculate anything for whether the total area initially covered by daisies exceeds what is possible. If this is the case, your code might spit out a warning message (a simple `disp` command would do). You might also terminate your program (see `exit`).

[20] Unless you have tested radically different areas for either white or black daisies.

Daisy population dynamics (1)

For an area fraction occupied by white and black daisies of α_w and α_b, respectively (note alpha here is fractional area, not albedo!), the change in occupied fractional area with time (t) can be written:

$d\alpha_w/dt = \alpha_w \cdot (x \cdot \beta_w - \gamma)$

$d\alpha_b/dt = \alpha_b \cdot (x \cdot \beta_b - \gamma)$

where x is the free (i.e. not occupied by daisies of any color) area of (fertile) ground, equal to:

$x = 1.0 - \alpha_w - \alpha_b$

(assuming here, unlike the more general case in *Watson and Lovelock* [1983], that all the land area is potentially fertile), β is a temperature-dependent growth function (one for each species of daisy), and γ the mortality rate (as a proportion of the area covered by that species of daisy per unit time). The value of γ given in *Watson and Lovelock* [1983] is 0.3, but this could be a parameter that you could play about with and investigate its effects.

To simplify things initially, temperature-dependent growth is a function only of the global mean temperature:

$\beta_w = 1.0 - 0.003265 \cdot (22.5 - T)^2$

$\beta_b = 1.0 - 0.003265 \cdot (22.5 - T)^2$

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 4.6.

The last step is given each species of daisy a different environmental preference for growth. We could simply given them different temperature optima, which is what the value of 22.5°C accomplishes in the temperature-dependent growth modifier equation. *Watson and Lovelock* [1983] take a different approach, and while assuming that both species of daisy have the same temperature preference, assume that they 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 4.7.

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.

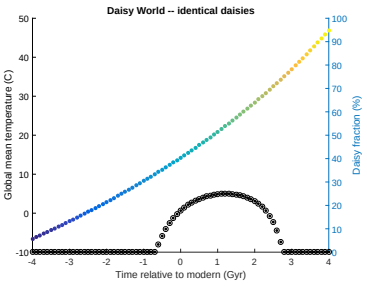


Figure 4.6: Evolution of global surface temperature and the two populations of daisies with time ... but now assuming that the growth of each depends only on the global mean surface temperature. Symbols as per Figure 4.5.

Daisy population dynamics (2)

To make the different species of daisies interact differently with the environment, the temperature-dependent modifiers of growth are made functions of the local (to the daisy population or individual), rather than global, temperature:

$\beta_w = 1.0 - 0.003265 \cdot (22.5 - T_w)^2$

$\beta_b = 1.0 - 0.003265 \cdot (22.5 - T_b)^2$

There are all sorts of says of defining how the local temperature deviates form the global mean. In *Watson and Lovelock* [1983] this is simply reduced to a simple deviation that scales linearly with the difference between mean global and local (daisy) albedo:

$T_w = T + q \cdot (A - A_w)$

$T_b = T + q \cdot (A - A_b)$

(noting that A is albedo here, not alpha as was the case in the original (non daisy enabled) EBM). q is a simple scaling factor that describes how strongly the local temperature deviates from the mean (or conversely, how efficiently heat energy is mixed between differen daisy fractions) and is assigned a default value of 10.0.

In the final zero-D EBM example, we'll make the model (very) slightly more interesting, or at least, (very) slightly more realistic. The time-dependent behavior of the simple energy balance model is trivial. In fact: there isn't any. The system is always in equilibrium as constructed. Why? No thermal inertia – i.e. nothing in the system defined so far has any heat capacity. So we need to add an ocean, or rather: a box (*variable*) to store the heat content, or temperature, of the ocean, and update this (temperature) in the event of there being

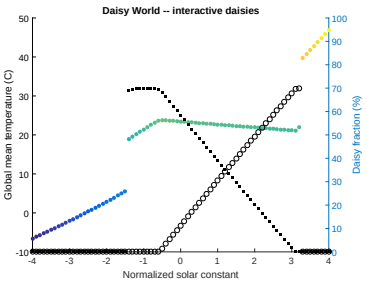


Figure 4.7: Evolution of global surface temperature and the two populations of daisies with time. Symbols as per Figure 4.5.

any imbalance between gain and loss of energy at the surface of the Earth. You can relate a temperature change to net energy gain (or loss) via the specific heat capacity of a substance (assuming water here).[21] You can also assume the following:

- The average mixed layer depth of the ocean is 70 m.
- The average fraction of the Earths surface that is ocean is 0.7.

(both from *Henderson-Sellers* [2014]).

Create yourself a new function, taking two parameters as inputs: (1) the total simulation duration, and (2) the time-step, both in units of yr.[22] You'll also need some of the constant values from before.

Break the code down into logical sections. Start by defining any constants you need, as well as parameter values (in addition to the two you have passed in). Then create a loop, which should repeat sufficient times to generate the requested simulation duration and at the requested time-step. Then plot something helpful at the end. In the loop itself, you firstly need to calculate the energy imbalance (assuming there is one), then use this flux to update the temperature of the mixed layer ocean. [23] If successful, you should see something similar to (actually, identical to) Figure 4.8 (assuming a 1 yr time-step). (Play about with the time-step in the model and note that as per previously, some care has to be taken with its choice, e.g. Figure 4.9 has a time-step of 3.5 years, which clearly is on the verge of going doolally).

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 CO_2, and radiative forcing due to the 'greenhouse effect', is just such an example.

The next example takes the time-stepping simple zero-D EBM that you created previously, and drives it with a time history of atmospheric CO_2 concentration (technically: mixing ratio) data.

First off: check out the CO_2 radiative forcing (Greenhouse Effect) Box. This will guide you as to modifying your energy budget (within the time-stepping loop). Test the model first with a fixed, assumed CO_2 concentration and check that the mean surface temperature responds in a reasonable way.[24,25]

[21] Once again – be very careful with the units. Or all will be lost ...

[22] So please don't forget that the energy flux has units of $Jm^{-2}s^{-1}$ (Wm^{-2}).

[23] It is much easier and less prone to bug, if you do this in two stages. You could even split things into four:

1. Incoming energy flux.
2. Outgoing energy flux.
3. Net energy flux at the Earths surface.
4. Update surface temperature.

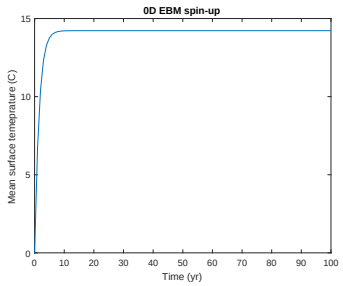


Figure 4.8: 100 yr spin-up of the basic EBM.

Doolally
Mad, insane, eccentric.

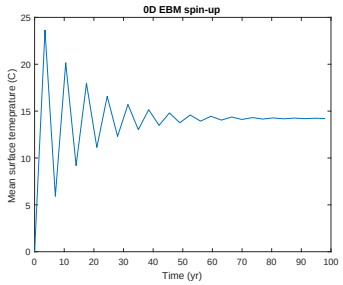


Figure 4.9: 100 yr spin-up of the basic EBM, but with a poor choice of time-step ...

[24] What is 'reasonable'? Well, you could conduct a pair of experiments – one in which you do not modify CO_2, and one in which your double it. The IPCC and there (now) five Assessment reports have much to say about the climate system response to a doubling of CO_2. So you can conduct a reality check on your model based on existing and widely available climate sensitivity information.

[25] By way of reference: assume that the pre-industrial concentration (mixing ratio) of CO_2 in the atmosphere ($CO_{2(0)}$) is 278 ppm.

You are going to load in a CO_2 data-set *externally* to the EBM function, and pass in an array of time (year) vs. CO_2. This is instead of passing in the experiment duration and time-step. In fact, you are going to determine the experiment duration and time-step according to the data ... So:

1. Firstly, get hold of the CO2 data. See: http://scrippsco2.ucsd.edu/data/atmospheric_co2 and download the spline fit version of the 'Merged Ice-Core Record Data'. You should get an Excel sheet for your trouble. Now you can load in Excel format data in MATLAB ... or simply copy-paste (into a text editor) the column of year and column of CO_2 data together. Load this into your workspace. Perhaps view the data in the **Variable Window** or plot it, just to be sure what you are working with.
2. Adjust the input parameters of your function so that you are just passing in the CO_2 data variable (consisting of n rows by 2 columns).
3. You need to know how many times to iterate through the loop, so creating a parameter and whose value you then set to the number of rows of data[26], is a good next step.
4. The time-step is an interesting issue. You could assume one and keep it at a fixed duration, or you could use the interval between CO_2 data points, as the time-step interval.[27] Take as the duration of each time-step, the time between successive each pair of data points.
5. The only minor complication is the CO_2 value to utilize – in updating the surface temperature between any two CO_2 data points, on average, the relevant radiative forcing is a function of the mean CO_2, rather than either one or the other of a pair.

(In summary: in calculating the change in temperature between year n and $n+1$, the time-step is the difference between these years, and the CO_2 value to use is the mean of the CO_2 values at year n and $n+1$ (because you can assume that CO_2 is continually changing).

When you have this working, try restricting the plot of mean annual global surface (air) temperature to year 1800 onwards. And perhaps restrict the y-axis range too. If you want to be fancy (e.g. Figure 4.10) you can draw on a horizontal line indicating the pre-industrial equilibrium solution (using the `line` function).

Finally, the lagged behavior of the climate system (as encapsulated in your EBM) is maybe not obvious as the forcing (CO_2) is varying. Nor would it necessarily help to cross plot the two, or plot both on the same plot, as radiative forcing has a log relationship with CO_2 change and temperature is not a simple function of radiative forcing

[26] Don't cheat! Determine the number of rows of data automatically.

[27] It happens that the data is evenly spaced at yearly intervals, but it need not have been.

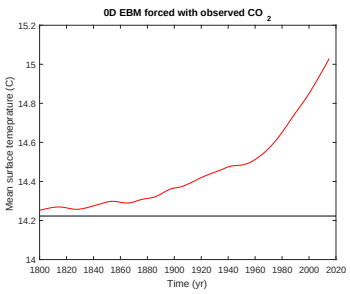


Figure 4.10: Transient EBM response to observed changes in atmospheric CO2. For reference, the pre-industrial equilibrium global temperature is shown as a horizontal black line.

`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.

(even at equilibrium). Common in model experiments and characterization, is to create artificial and deliberately simplified forcings and perturbations, so as to more readily diagnose the response time and characteristics of a system. Crete an artificial CO_2 data-set, spanning the same time interval as the real data, and at the same frequency, but substitute an idealized CO_2 forcing in which CO_2 stays constant (at 278 ppm) up until year 1999, then at year 2000, increases to 400 ppm, and stays there. This should look like Figure 4.11.

You could go on to test instantaneous doublings, and quadruplings of CO2 – both classic and commonly-used perturbations. Also common are linear ramps (up, and/or down) and compound increases, such as a 1% per year increase in the concentration of CO_2 (each and every year) starting ca. 1960.

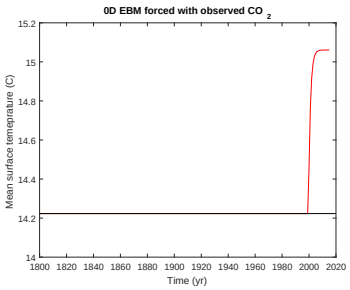


Figure 4.11: Transient EBM response to (fake) changes in atmospheric CO2.

The Greenhouse Effect

The effect of changing CO_2 concentrations on the global energy budget is typically written in terms of a virtual (long-wave) radiation flux applied at the top of the atmosphere. The flux anomaly, ΔF, as a function of CO_2 concentration (technically: mixing ratio) (CO_2) relative to a reference (pre-industrial) concentration ($CO_{2(0)}$) can be approximated:

$$\Delta F = 5.35 \cdot ln(\frac{CO_2}{CO_{2(0)}})$$

5
1- and 2-D numerical modelling

5.1 1-D models

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 varies vertically much quicker than it does horizontally (why you can suddenly get snow 6000 ft up in Idyllwild, but not just by going 2 miles to the North of downtown Riverside at sea-level (ish)). 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 with 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 sometimes be analysed by considering their behaviour in just one dimension.

The simplest possible[1] 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 Earths surface 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 Earths surface up into latitudinal bands. Why latitude and not longitude? Simple inspection of global temperature distributions (or other climate properties such as wind fields) indicate that the meridional[2] gradients are much more pronounced that the zonal[3] gradients. Obviously, a model would be improved by resolving both

[1] :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**

[2] According to the mighty Wikipedia: "along a meridian" or "in the north-south direction".

[3] "along a latitude circle" or "in the west-east direction"

meridional and zonal gradients and energy flows, but if you are going to simplify an energy balance model to just one dimension – latitude is it. You can also think in terms of how incoming solar radiation changes most – ignoring day-night changes as the Earth rotates – the Equator vs. poles has the greatest contrast in incoming energy, 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).[4,5] So, for this exercise, you need actually only model one hemisphere (and assume that the other one acts 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, which each band being subject an an energy budget including an ocean-dominated heat capacity and 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 as before.) You can chose how many bands to make. Actually, if you do it the 'easy' way it will not matter how many you want[6] and which, as you might have guessed, uses loops. The hard way is to write out all the equations explicitly[7]. 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[8]. For each band, you can write exactly the same equations as before. Except ... for the in-coming solar radiation. Why? (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.) So you need to calculate the surface area of each band, assuming that each band occupies an equal number of degree of latitude, and how this varies with latitude? A small hint can be found in Box #1. Or the Internet will, as usual, know all.

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) 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

[4] 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.

[5] 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.

#1 Zonal area of the Earths surface

The area of a zonal band of the Earth surface, from latitude ϕ_1 to ϕ_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:

$\sum_{\phi_1}^{\phi_2} 2 \cdot \pi \cdot r_0 \cdot cos(\phi) \cdot \delta x$

where δx 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 $r_0 \cdot \delta\phi$. Hence, in the lim $\delta\phi \rightarrow 0$:

$\int_{\phi_1}^{\phi_2} 2 \cdot \pi \cdot r_0^2 \cdot cos(\phi)\ d\phi$

The zonal area between latitude ϕ_1 and ϕ_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°to +90°(or -$\pi/2$ to +$\pi/2$) you recover the surface area of a sphere: $4 \cdot \pi \cdot r_0^2$).

[6] 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.)

[7] 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.

[8] 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.

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 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 ° 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 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`'[9] 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 5.1. 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[10] of some sort to be sure the model projection is not bonkers. You had

#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, ψ:

$A = \frac{r_0^2}{2} \cdot (\psi - sin(\psi))$

Again, you need a picture. If we re-write ψ in terms of latitude ϕ:

$\phi = \frac{(\pi - \psi)}{2}$

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.

[9] 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).

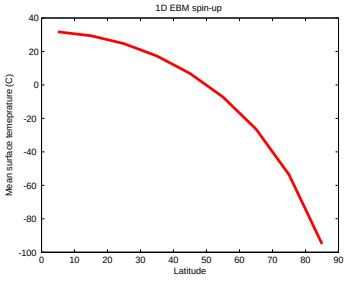


Figure 5.1: Basic 1-D EBM with no latitudinal heat transport.

EXAMPLE OVERVIEW:

1. **Load, process, and overlay observations**

[10] Not the Star Trek, Next Generation, one.

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 mean. Actually, **MATLAB** makes this obscenely simple for you using the `mean` function[11]. 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.

Once you have fixed any obvious data problems, you should end up with something like Figure 5.2 (January) or Figure 5.3 (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 a 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.[12]

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

[11] A function to calculate the arithmetic mean, rather than a nasty and vindictive function.

`mean`

MATLAB `help`, helpfully says:

```
Average or mean value.
S = mean(X) is the mean
value of the elements in X
if X is a vector.
For matrices, S is a row
vector containing the mean
value of each column.
```

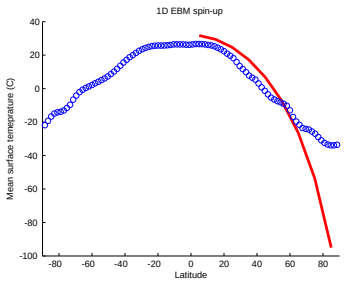


Figure 5.2: Basic 1-D EBM with no latitudinal heat transport (red filled circles). Overlain is the zonal mean observational data for January (blue circles).

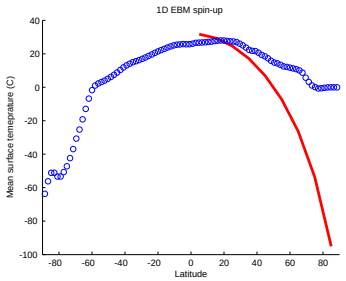


Figure 5.3: As per Figure 5.2 but for July.

[12] 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.

EXAMPLE OVERVIEW:

1. **add heat diffusion term**
2. **(manually (i.e. 'trial-and-error')) optimize model**

up with similar looking equations. Taking the thermal conductance approach, the flux of heat energy from one latitudinal band to the next, J (W), can be written[13]:

$$J = -k \cdot A \cdot \frac{\Delta T}{\Delta z}$$

where k is the thermal conductivity ($Wm^{-1}K^{-1}$), ΔT is the difference between the temperatures of two adjacent zonal bands ($T_1 - T_2$), and Δz the distance between the bands (measured at the mid-point latitude[14]). This is effectively the same as for the diffusion of CH_4 in a soil column, 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 cm^2) basis and hence area did not appear explicitly in the equations. 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 your understanding(!) of the CH_4 model – 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:

$$\Delta F_{(n)} = F_{solar\_in\,(n)} - F_{longwave\_out\,(n)} + F_{diffusion\_in\,(n)} - F_{diffusion\_out\,(n)}$$

(This is all going to very much end up looking quite like the CH_4 diffusion model, in that we have special boundary conditions to consider for the zone bordering the Equator and the zone bordering the pole, 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. For the lowest latitude zone, if we are assuming that the

[13] The equation is conventionally written as negative, assuming the point of reference is the higher temperature, which loses heat energy.

[14] 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(!)).

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, ϕ (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).

Earth is symmetrical about the Equator, then it only loses heat to a higher latitude zone and does not gain heat from anywhere.)[15]

The structure of your model, within the (outer) time-stepping look, should then look like:

1. Loop through all n latitude bands and calculate the in-coming and out-going radiation.[16]
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[17] ... and adjust your guess if the model fits the data worse than before.

As an illustration – Figure 5.4 shows the effect of specifying a value of heat conductivity of $k = 0.1\ Wm^{-1}K^{-1}$, while $k = 1.0\ Wm^{-1}K^{-1}$, as shown in Figure 5.5, is clearly compete 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 Souther to North.)

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 (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 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

[15] 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.

[16] Don't update any temperatures yet!

[17] 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.

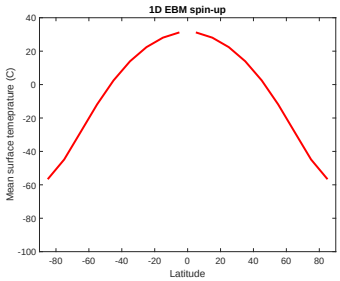


Figure 5.4: 1D EBM with an initial guess as to the value of k.

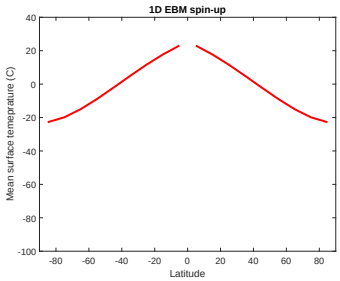


Figure 5.5: 1D EBM with a x10 larger value of k.

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

in this exercise in any role that the soil methane sink might play in controlling the concentration of 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 CH_4 (there will also thus also be a return of CO_2, the metabolic product of CH_4 oxidation, from the soil to the atmosphere). Because CH_4 is being depleted at depth, there will be a gradient in CH_4 concentrations along which CH_4 there will be net diffusive transport, illustrated in Figure 5.6. The scientific question is thus; what is the flux of 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. [18]

If all 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 CH_4 (in soil), with units of $cm^2 s^{-1}$, then we can easily calculate the initial flux, J, of methane into the soil by Fick's law (as cm^3 CH_4 per second (s^{-1}) per unit cross-sectional area (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 CH_4 concentration (i.e., the change in concentration divided by the change in depth).

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_4 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_4 in the soil column, illustrated in Figure 5.7. We will assume an initial value for D of 0.186 cm^2 s^{-1}.

Because we would quite like a general model for soil CH_4 uptake that was capable of accounting for these sorts of complications if nec-

[18] In reality the system looks more like Figure 5.7, and actually, even more like Figure 5.8 ... adding considerable complexity (and dynamics).

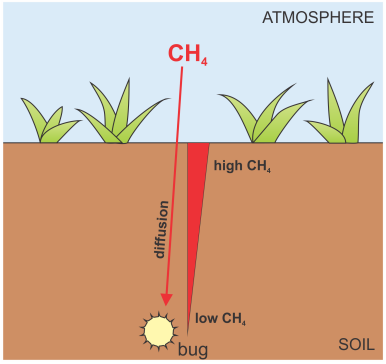


Figure 5.6: Idealized schematic of the soil-CH4 system.

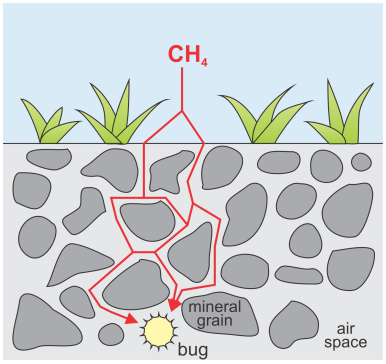


Figure 5.7: Slightly less idealized schematic of the soil-CH4 system.

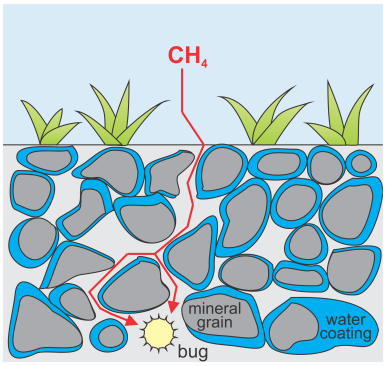


Figure 5.8: Even less idealized and almost realistic, schematic of the soil-CH4 system.

essary, 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_4 diffusion into the soil, although there is not time in this exercise to go on and consider the metabolic consumption of atmospheric CH_4 by methanotrophic bacteria.

If we divide up the soil profile into 10 equally-spaced (equal thickness) layers[19], the basics of the model will be an array with 10 rows, one (row) location in the array representing the CH_4 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, C1, #2 corresponds to the 0-1 cm depth interval, C2, ... , and #10 corresponds to the 9-10 cm depth interval, C10. We will also need to create an array to store the average depth, z_n at which each of the CH_4 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}).

[19] It need not be 10 – choosing 10 layers of 1 cm thickness each, just simplifies things.

We are now ready to calculate the diffusion of CH_4 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_4 between 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[20,21,22] – 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_4 in soil pore spaces rather than metals in solution in a lake, and we have diffusion of CH_4 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_4. 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_4 between layers and

[20] Except less wet.

[21] And smaller.

[22] And in the soil ... OK, so not so much like the Great Lakes system ...

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*.[23]

2. Create a 10×1 vector array call `conc` and initialized with all zeros[24]. This is the variable array for storing the concentration of CH_4 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_4 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_4 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_4 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_4 in the atmosphere. To keep things as simple as possible, you will be assuming units of $cm^3\ cm^{-3}$, so that the atmospheric CH_4 concentration becomes $1.7\times10^{-6}\ cm^3\ CH_4\ cm^{-3}$ (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 ($0.186\ cm^2\ s^{-1}$):

```
D = 0.186;
```

3. Create a basic time stepping loop. Define a time-step length (`dt`) to take – this is the amount of time that going around the loop each time represents. Call the time-step length parameter `dt` 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

[23] 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 ... :(

[24] To save time – use the **MATLAB** function `zeros`.

top of the script). The model simulation length you want is given by the (passed) parameter `maxtime`, and each time around the loop lasts `dt` 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()`[25]) the current loop value of `tstep` as well as the time elapsed in the model.[26] 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
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.[27]

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_4 in the soil column – you will

[25] The display line(s) should go inside the loop, of course.

[26] Equal to the loop count multiplied by the time-step length:

```
tstep*dt
```

[27] 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.

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 2.0×10^{-6} cm^3 cm^{-3}. Also plot on the same graph as a point the atmospheric CH_4 concentration at the surface of the soil, whose value is held in the parameter `conc_atm`.[28,29,30]

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^3 cm^{-3}, together with a point at the top (depth = 0.0) marking the atmospheric CH_4 concentration of 1.7×10^{-6} cm^3 CH_4 cm^{-3} (or 1.7 ppm if you have re-scaled the x-axis values). Check that you have this. Note that the CH_4 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_4 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_4 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_4 between each layer. The concentration units of CH_4 are cm^3 CH_4 cm^{-3}. The length scale is cm. The diffusivity of CH_4, D has units of cm^2 s^{-1}. So if we apply dimensionality analysis (basically, just working out the net units) we get:

$$J = cm^{-2} \times \ cm^3 \ \ CH_4 \ \ cm^{-3}/cm$$

which comes out to give J in units of cm CH_4 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 cm2 (i.e., 1cm $\times$ 1 cm). Therefore, the flux of CH_4 is occurring in a 1 cm^2 unit cross sectional area model, with units of:

$$J = cm^{-2} \times \ cm^3 \ \ CH_4 \ \ cm^{-3}/cm \times \ cm^2$$

or cm^3 CH_4 s^{-1}. This is much more reasonable (and cm^3 of CH_4 can easily be converted into units of moles or g of CH_4 if you needed to).

[28] `hold on` and then using the `scatter` function is probably the easiest way.

[29] 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 ...)

[30] 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 10^6 times bigger; i.e.:

```
plot(1.0E6*conc(1:10),-z_mid(1:10));
axis([0 2.0 -10 0]);
```

9. Before adding in the meat of the model (the calculation the fluxes of CH_4 between the pairs of 1 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 (Δz) over which diffusion occurs is only 0.5 cm in this case (measuring from the surface (zero height) to mid-depth of the first 1 cm thick layer). Referring to the equations previously, but recognizing that the $n = 0$ layer doesn't exist because it is the atmosphere[31] (so `conc(0)` and `z_mid(0)` have been replaced by `conc_atm` and `z_atm`, respectively) you should see that the flux of CH_4 into the first soil layer from above is:

```
J(1) = D*(conc_atm - conc(1))/(z_mid(1) - z_atm);
```

[31] And also because you cannot start indexing a vector in **MATLAB** at zero.

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 CH_4 into layer n from the layer above ($n - 1$) can be written:

```
J(n) = D * (conc(n-1) - conc(n)) / (z_mid(n) - 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.[32]

[32] Don't forget that you have just calculated the first $n = 1$ layer flux (`J(1)`) already.

11. Make sure that you are happy with what you have done so far. You have calculated the CH4 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!).

Although you are not yet updating the concentration of CH_4 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_4 in each soil layer and see what happens ... To keep it simple, assume that the soil has a porosity of 1 cm^3 cm^{-3} (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_4 concentration in soil layer n due to the flux of CH_4 from above (layer $n - 1$) you must add a volume of CH_4, given by the calculated J_n value (in cm^3 of CH_4 per second) multiplied by the time-step interval (in s). You must also take into account the loss

of CH_4 from each soil layer n as CH_4 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;

```
conc(1) = conc(1) + dt*J(1) - dt*J(2);
```

This is saying that the new CH_4 concentration in layer $n = 1$ is equal to the concentration at the previous time-step, plus the CH_4 that diffuses into the later from above (`J(1)`), minus the CH_4 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_4 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_4 concentrations for layers $n = 1$ through 9:

```
conc(n) = conc(n) + dt*J(n) - dt*J(n+1);
```

Now add in the code to update the $n = 10$ layer CH_4 concentration (i.e., adding just the flux from above (`J(10)`) to the current `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 CH_4 concentrations as you go down deeper into the soil, looking something like Figure 5.9.

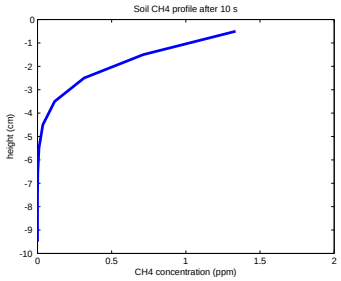


Figure 5.9: Soil profile of CH_4 after 10.0s of simulation.

Now try a longer model run (100 s) (»`ch4model(100.0)`) and see what happens. You should get something like Figure 5.10.

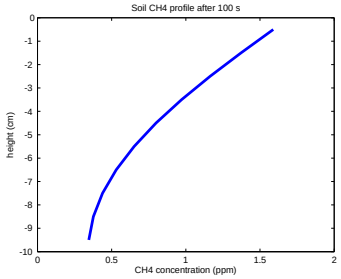


Figure 5.10: Soil profile of CH_4 after 100.0s of simulation.

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.

Keeping with the same Example[33] and having constructed the basic diffusion framework for the model, we can explore what happens if consumption of CH_4 (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`).[34] 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_4 profile that lead you to suspect instability occurring in the numerical solution? The onset of instability might look something like Figure 5.11.

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_4 that diffuses down from the

[33] 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_4 sinks)
4. play!

[34] 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.

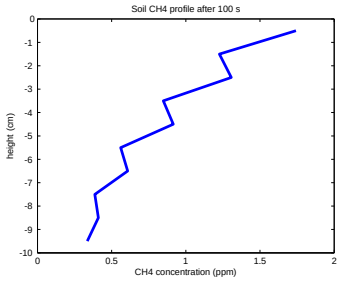


Figure 5.11: Soil profile of CH_4 after 100.0s of simulation with an extremely marginal choice of time-step length.

atmosphere into the soil pore-space. A bit like idle grad students living on a diet of pizzas.[35] The bugs consume the CH_4 present in the soil pore space at a rate that is proportional to the concentration of CH_4 in the soil (makes sense – the more CH_4 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_4 in layer $n = 10$ is `conc(10)`, then in one second:

[35] Except students mostly don't live in the cold damp dirty ground.

```
munch_rate * conc(10)
```

cm^3 CH_4 cm^{-1} will be lost from the soil pore space. So, if you had a rate constant (`munch_rate`) of 0.5 s^{-1}, then each second, half of the CH_4 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_4 each time around the loop would be:

```
0.1 * munch_rate * conc(10)
```

cm^3 CH_4 cm^{-1}. Are you following so far ... ?

Now, add a third parameter that is passed into the soil CH_4 model function for the rate constant. Modify your equation for the updating of the CH_4 concentration in the deepest (n=10) soil layer to reflect the presence of the methanotrophs. Call the soil CH_4 model function; pass a time-step of 0.1 s and a methanotroph CH_4 consumption rate constant of 1.0 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[36,37]. 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 CH_4 concentration, and why?

[36] Not your favourite website address.

[37] e.g. for 100s, giving a plot looking (hopefully) like Figure 5.12.

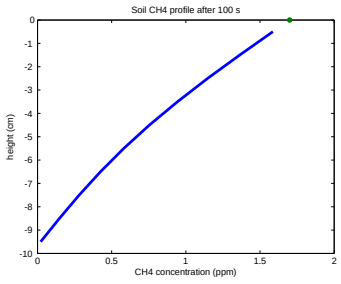


Figure 5.12: Soil profile of CH_4 after 100.0s of simulation, with CH_4 uptake at the base of the profile with a rate constant of 1.0 per s.

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 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 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 CH_4 concentrations) to reflect the consumption of 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 s^{-1}, rather than the value of 1.0 s^{-1} that you used before) for all subsequent calculations. Call the soil CH_4 model function with a time-step length of 0.1 s and determine the steady state soil (equilibrium) CH_4 profile (Figure 5.13). What shape does this remind you of ... and why?[38]

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}[39] (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 CH_4 at the base of the soil column.[40] [41] Units should be: cm^3 CH_4 cm^{-3} 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 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_4 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_4. To do this – you need to extract from the model, the CH_4 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[42] – e.g. Figure 5.13.

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_4 *to* the atmosphere. Can you find this value (which makes the net exchange zero) ... *without* trial-and-error?[43]

[38] There is in fact an analytical solution to this profile – can you derive it?

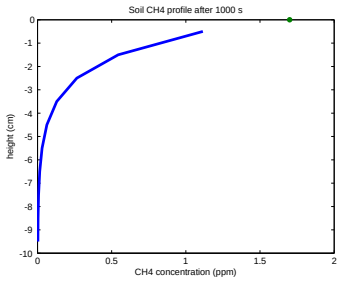
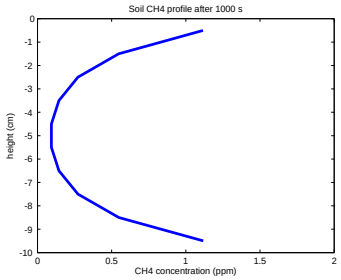


Figure 5.13: Equilibrium soil profile of CH_4, with CH_4 uptake throughout the soil column with a rate constant of 0.1 per s.

[39] For which you might call the variable, e.g. `n_max`).

[40] 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.

[41] 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.



4

Figure 5.14: Example equilibrium soil profile of CH_4 with production at depth.

[43] 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.

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.

Bibliography

Stormy Attaway. *Matlab (Third Edition): A Practical Introduction to Programming and Problem Solving*. Butterworth-Heinemann, 2013.

Index

... environment, 60
.mat environment, 32
; environment, 21
= environment, 20, 21

addition environment, 21
addpath environment, 28
and environment, 22
assignment operator environment, 22
axis environment, 34

cell array environment, 31
cell2mat environment, 31
clear all environment, 23
clear environment, 23, 38
close environment, 23
colon operator environment, 23, 24
colorbar environment, 40
Command Window, 17
comment environment, 28
comments environment, 30
contour environment, 43
contourf environment, 43

disp environment, 50, 53
division environment, 21

else environment, 60
elseif environment, 60
environments
  ..., 60
  .mat, 32
  ;, 21
  =, 20, 21
  addition, 21
  addpath, 28
  and, 22
  assignment operator, 22
  axis, 34
  cell array, 31
  cell2mat, 31
  clear, 23, 38
  clear all, 23
  close, 23
  colon operator, 23, 24
  colorbar, 40
  comment, 28
  comments, 30
  contour, 43
  contourf, 43
  disp, 50, 53
  division, 21
  else, 60
  elseif, 60
  equality, 22
  exist, 61
  exit, 23, 86, 87
  exponentiation, 21
  figure, 33
  find, 38
  fliplr, 25
  flipup, 25
  fopen, 30
  for, 52
  fprintf, 32
  functions, 22
  geoshow, 47
  getframe, 56
  greater than, 21
  greater than or equal to, 21
  Headerlines, 31
  hist, 42
  hold, 36
  icecream, 67
  if ... end, 60
  image, 43
  inequality, 22
  legend, 37
  length, 58
  less than, 21
  less than or equal to, 22
  line, 90
  load, 28, 32, 35
  ls, 28
  m-file, 49
  m-files, 34
  mean, 96
  meshgrid, 45
  movie2avi, 56
  multiplication, 21
  NaN, 38
  not, 22
  num2str, 53, 54
  or, 22
  pi, 22
  plot, 33
  print, 35
  quit, 86
  rotate, 25
  save, 32, 75
  scatter, 38
  sin, 37
  size, 33, 66
  sort, 36
  sortrows, 36
  subplot, 37
  subtraction, 21
  sum, 25
  textscan, 30, 31
  title, 34
  transpose, 25, 33
  while, 52
  xlabel, 34
  ylabel, 34
  zeros, 66
equality environment, 22

exist environment, 61
exit environment, 23, 86, 87
exponentiation environment, 21

figure environment, 33
find environment, 38
fliplr environment, 25
flipup environment, 25
fopen environment, 30
for environment, 52
fprintf environment, 32
functions environment, 22

geoshow environment, 47
getframe environment, 56
greater than environment, 21
greater than or equal to environment, 21

Headerlines environment, 31
hist environment, 42
hold environment, 36

icecream environment, 67
if ... end environment, 60
image environment, 43
inequality environment, 22

legend environment, 37
length environment, 58
less than environment, 21
less than or equal to environment, 22
license, 2
line environment, 90
load environment, 28, 32, 35
ls environment, 28

m-file environment, 49
m-files environment, 34
mean environment, 96
meshgrid environment, 45
movie2avi environment, 56
multiplication environment, 21

NaN environment, 38
not environment, 22
num2str environment, 53, 54

or environment, 22

pi environment, 22
plot environment, 33
print environment, 35

quit environment, 86

rotate environment, 25

save environment, 32, 75
scatter environment, 38
sin environment, 37
size environment, 33, 66
sort environment, 36
sortrows environment, 36
subplot environment, 37
subtraction environment, 21
sum environment, 25

textscan environment, 30, 31
The command line, 17
title environment, 34
transpose environment, 25, 33
typefaces
  sizes, 77

variable, 18

while environment, 52

xlabel environment, 34

ylabel environment, 34

zeros environment, 66