ANDY RIDGWELL

str=’Do you like bananas?’;

UC-RIVERSIDE / DEPT. OF EARTH AND PLANETARY SCIENCES 2020/21

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

The last time anyone could be bothered to hack this document, was September 30, 2024

Contents

List of Figures

List of Tables

How to use this Textbook

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

0.1 Fonts and highlighting

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

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

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

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

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

or writing it out 'normally':

a = 10 x b + c^2

0.2 Help(!) and keyword definitions

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

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

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

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

0.3 Side notes and other distractions from the main text

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

[1] I am a Side note!
[2] I am also a Side note!

[3] Some are trivial and a little worthless educationally, but you wont know which is which until you have read them ... They might also just brighten up your day a little.

0.4 What and when to type

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

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

When the given examples are illustrating instructions typed in at the command line, the text again appears in the 'Typwriter' font, but in addition, the command line prompt (») is shown at the start of a line (you do not actually type in the prompt itself ...), e.g.

```
» hello
```

is typing in `hello` at the command line, and

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

is then showing you what happens (when you type in `hello` at the command line).

Additionally ... lines of code that go along with the discussion in the text and which are not necessarily intended for you to type in (although you may still want to, simply to try it out), are given in a light Courier font:

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

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

```
» disp('hello')
```

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

```
% place in a file
```

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

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

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

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

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

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

Alternatively:

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

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

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

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

0.5 Code structure

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

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

For the *function*, that information is passed into the *function*, and then returned back to where the function was called from, is indicated by the red arrows entering the top of the box and leaving the

[4] Not all code fragments and programs are given a schematic.

[5] Don't worry about the terms *function* and *script* for now ....

bottom of the box, respectively. (But note that there is no line of code at the end that tells the model to return values ... this is simply to illustrate the flow of the program, particularly when things get more complicated and there are multiple *scripts* and *functions* involved.)[6]

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

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

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

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

[6] All this should hopefully all become apparent later.

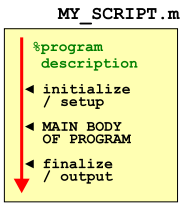

Figure 1: Schematic for a generic *script*.

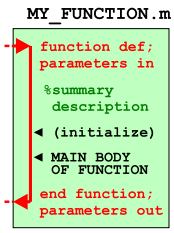

Figure 2: Schematic for a generic *function*.

[7] Also surrounded by flashing neon lights.

0.6 'Answer' codes

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

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

0.7 *MATLAB versions*

The **MATLAB** software suite is constantly evolving and new and/or improved functions are constantly appearing. The point of this course is to provide you with a basic programming ability, as well as practical skills in applying **MATLAB** to science and data problems, rather than skill you specifically and more narrowly in the most

recent **MATLAB** software version. You may well find 'easier' ways of doing things using newer **MATLAB** functions as compared to how things are described in this text – if you do, and want to utilize newer/simpler ways of doing things – then please feel free to do so!

1
Elements of ... MATLAB and data visualization

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

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

Note that for now (this Chapter), you will only be entering code at the *Command line* (not in a file, which will come later in Chapter 2).

22 str='do you like bananas?';

1.1 Using the MATLAB software

1.1.1 Starting MATLAB

To start with: find the **MATLAB** icon on the desktop; run the program. You should see a number of sub-windows arranged within the main **MATLAB** window, hopefully including at the very least, the *Command Window*[1]. Depending on whether you have used **MATLAB** before and it has remembered your settings, windows may also include: *Command History*, *Workspace*, *Current Folder*. If instead you see; 'Tetris', 'Grand Theft Auto: San Andreas', and 'Fortnite Battle Royale', then you have the wrong software running and are going to find learning **MATLAB** rather hard. However, there is big $$$ to be made in on-line gaming tournaments these days. You could quit your degree now ... Conversely, there are also good jobs if you are able to program ... so maybe stick with the course and read on ...

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

1.1.2 The command line

When **MATLAB** initially starts up, the *Command Window* should be blank except for a vertical blinking line (cursor) following the double 'greater than' symbols with an ocean of blank lines/space below[2,3].

If you are unfamiliar with using command-line driven software ... Don't Panic![4] Nothing bad can happen, regardless of what you do. Well, almost. It is possible to accidently clear **MATLAB**'s memory of the results of calculations and data processing and close plots and graphs before you have saved them, but **MATLAB** remembers all the commands you type, so in theory it is possible to quickly reproduce anything lost. (Later on (not now!) we will be placing the sequence of commands into a file (that is saved) and so ultimately, **MATLAB** should turn out to be mostly fool-proof.)

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

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

Well ... not so exciting. But not so disastrous! **MATLAB** simply has no clue what you are talking about, or rather, `anything` is not a 'key word'[5] that **MATLAB** recognises. In the specific error message, **MATLAB** could not find that `anything` was a built-in (or user-defined) *function*, nor a listed *variable*, both of which you'll learn about in due course.

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

[3] Older versions of **MATLAB** might have displayed:

```
Academic License
»
```

or even:

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

[4] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Pocket Books, 1979. ISBN 0-671-46149-4

[5] i.e. a word, or sequence of characters that has a special meaning to **MATLAB** and it will act upon, as opposed to a sequence of characters that has not special meaning and **MATLAB** completely ignores.

1.1.3 *MATLAB GUI*

There are lots of fancy looking icons and pretty colors and you could spend all day staring at them and not getting any work done. Or you could learn some good programming practice. Which is why we mostly will ignore the eye-candy and little (if any) guidance will be given as to the functionality of the Graphical User Interface (*GUI*). Look at this as a lesson for the user (to read the Help, textbook, online documentation, or simple go **Google** for an answer[6]).

[6] Otherwise known as Internet fishing.

1.1.4 *Help(!)*

If stuck at any point – you can press the `F1` key or click on the question mark icon on the tool-bar, to bring up the indexed and searchable **MATLAB** documentation.[7]

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

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

```
» help
```

The result is perhaps not especially helpful. The typical usage is to provide the name of a *function*[8] you require help on. Perversely, `help` is a *function* and **MATLAB** provides help on `help`. The initial output to which is as follows:

[8] Don't worry about what a *function* is yet.

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

In the course text, for each *function* that **MATLAB** provides a comprehensive help on, such as `help`, a simple summary version will be displayed in the right hand margin in a grey box.[9]

[9] Refer to the section on 'How to use this Textbook'.

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

24 str='do you like bananas?';

1.2 Basic concepts

1.2.1 Variables

A *variable* is, in a sense, a pointer to a location in computer memory where a piece of information is stored[10]. For instance – open up a blank worksheet in **Excel**, and in the very top left hand cell, enter the number `10`. You can see visually, that **Excel** is referencing this location as column A, and row '1'. In fact, this location ('A1') is indicated in the Name Box to the left of the Formula Bar.

In **MATLAB** (and other programming languages), a variable (or rather, the value of a variable) is associated with a name (rather than a letter-number code as in **Excel**) in order to make things rather more easy and convenient. The name can be almost any sequence of characters you like, regardless of whether it is a real or fake word, just as long as it does not contain numbers or special characters (e.g. #, $, %, ...) or spaces. So actually, you are only left with continuous sequences of characters without spaces ('words'!). Note that you can create a variable name based on two (or more) real words, separated by an underscore (_) if that helps describe what the *variable* refers to. Valid variable names include:

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

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[12].

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

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

- **Integer** – An integer number is a counting number, i.e. `1, 2, 3, ...` and including zero and negative integers.
  (**MATLAB** has different representations for integer numbers, depending on how large a number you need to represent (and how much memory it will need to allocated to storing it). This is something of a throw-back to the days when computers only

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

[11] 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). But other programming languages may not.
I would strongly advise to stick to all lower case (or all upper case), to avoid possible future confusion. (or come up with a naming convention, of whatever sort (e.g. capital first letter), and stick to it.)

[12] 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.

had $1/10000000^{th}$ of the memory of your iPhone and were slower than half a lemon nailed to the floor. So we will not, in this text, particularly worry about a numbers/computing concept called *precision*.)

- **Real (floating point)**[13] – 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. (But as per for *integers*, we will not worry about this in the course.)

  Be aware that you can configure[14] **MATLAB** to display a particular format for real numbers, e.g.

  `42.0`
  versus
  `4.2e+01`

  These are identical *real* numbers, just each with a different display format).

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

  Related to this is the **String** *type*, which is more flexible esp. for creating arrays of characters. Which then confuses all the description in the text as a sequence of characters (or a *vector* of characters) has since the dawn of time been known as a *string*. In the text, *string* will be used to refer to *vector* of characters rather than the **MATLAB string** type.

  (Confused already? It will all become clearer with practice ...)

- **Logical** – a *variable* that can be `true` or `false`[15] – we'll come to quite what this means later.

- **etc** – No, not a real *type*, but to note that **MATLAB** defines and recognises a whole bunch of other *variable types*, including **Complex** (**MATLAB** can handle *complex numbers*) and **Object** (we will also not worry about *objects*, which can incorporate a combination of types. At least, not yet ...). The **MATLAB** documentation contains a full list (and/or go Internet Fishing).

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

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

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

[14] Under the menu item Preferences and then Command Window.

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

avoided. However, even in **MATLAB**, *strings* and *reals* (or *integers*) are very different things.[16] When necessary, different *variable types* can be converted between (see **Variable Type Conversion** Box).

The second and perhaps rather more important thing, is how to assign a value to a *variable* (and in fact, create the variable in the first place). Programming languages such as **FORTRAN** require you to define the variable beforehand and assign it a *type*.[17] **MATLAB** allows you to define and assign a value to a *variable* all at the same time, and it will kindly work out the correct *type* based on the value you assign to it.

You assign a value to a *variable* using the *assignment operator* =[18]. For example:

```
A = 10
```

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

```
A =
10
```

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

```
A = 10
```

is interpreted exactly the same as:

```
A=10
```

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

MATLAB will also report in the `Workspace` window, the name and value, *type* (unhelpfully called *Class*), etc of all your current *variables* (just one currently?). Actually, it is not all quite so simple. If you take a look at the *Class* of the *variable* `A` in the display window – it is listed as *double* (a *real* number) rather than an *integer*. So by default, if **MATLAB** does not know what you really want, it defines `A` as a double precision real number[19].

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

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

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

[17] Partially true. An Alternative Fact of sorts.
[18] This is NOT 'equals' in **MATLAB**. Or any sane programming language. We will see the *equality operator* shortly. = assigns the value or variable on its right, to the variable on the left.

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

Pausing again ... if you want to remind yourself of the *variables* that you (or a program) have created – you can refer to the Workspace window.[20] Also listed here as noted above, is its value (and *type* etc). Another way to access the value of a *variable*, is to simply type in its name at the command line:

```
» A
```

and **MATLAB** will parrot back:

```
A =
10
```

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

```
B = apple
```

and **MATLAB** is far from happy. As it turns out, a sequence of characters can also refer to a *function*[21] in **MATLAB**, and this is what **MATLAB** looks for – i.e. a match to `apple` in the list of *variable* (and *function*) names. In other words, **MATLAB** does not know whether you intend `apple` to be a *string* or a *function*. It assumes *function* ... but cannot find one with that name and then gives up.

To delineate `apple` unambiguously as a *string*, you need to encase it in (single or double) quotation marks:

```
B = 'apple'
```

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

```
A = 'banana'
```

has now replaced the real number `10` in variable `A`, with the character string `banana`. This is reflected in the updated variable list details given in the Workspace window (and the variable *type* or *Class* is now listed as `char`).[22]

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

```
C = 'totalbanana';
```

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

[20] There is a command line command for listing current variables (`whos`), but lets not bother with it.

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

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

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

1.2.2 *Numerical expressions and Arithmetic operators*

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

- `exponentiation` — ^ — 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` — × — e.g. a×b, written in **MATLAB** as `a*b`.
- `division` — / — (written as you would expect).[24]
- `addition` — + — (guess).
- `subtraction` — - — again, obvious/intuitive.

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

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

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

```
B = -A
```

The assignment operator (=)[25] comes last.

[25] This is **NOT** 'equals to'.

If you are unclear about the order numerical operators are carried out, then place parentheses ( ) around the component of the calculation you wish to be carried out first to enforce a particular order (this can also help in making an equation easier to read and ultimately, easier to debug code). For example, consider:

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

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

1.2.3 *Relational and logical operators*

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

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

and test the relationship between 2 variables.

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

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

```
A =
     1
```

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

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

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

1.2.4 *Functions (built-in)*

MATLAB provides numerous built-in *functions*[26]. These *functions* have specific names assigned to them, so care needs to be take not to give a *variable* the same name as a *function* to avoid getting confused further down the road. Giving an exhaustive list (and brief description) is outside the scope of this text[27]. Common *functions* will be progressively introduced as this text progress. Note that in addition to the on-line `Help` documentation, information on how to use a *function* and example uses is provided by typing `help` and then the *function* name (separated by a space) at the command line.

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

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

MATLAB also provides several built-in mathematical *constants* (which save having to define a variable with the appropriate number that you no-doubt will have to look up from the internet first ...). This are simply *variables* that have been already defined and assigned values, but which you cannot change (hence 'constant'). For instance, the value of π is assigned to a built-in *function* with the name `pi`. You can recover its value by typing its name at the command line:

```
» pi
ans =
    3.1416
```

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

1.2.5 Miscellaneous commands

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

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

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

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

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

[28] An 'alternative' pi?

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

1.3 *Vectors and arrays #1*

The *variables* that you have seen so far are known as *scalars* – i.e. single numbers (whether *real* or *integer*)[30]. One of the most powerful things about **MATLAB** is its ability to represent vectors (1D columns or rows of numbers or characters) and arrays – 2D and higher dimensional regular grids of numbers or strings. (*matrix*[31] is the name commonly given to a 2-D array.)

[30] An exception are when you assigned a string, which technically is a vector (assuming multiple characters in the string)

[31] Not to be confused with the film containing bad acting by Keanu Reeves.

1.3.1 *Creating vectors*

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

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

or with the values comma-separated:

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

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

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

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

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

which gives the maths-speak representation:

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

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

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

```
A = j:k
```

and

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

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

```
» A = 0:3
A =
     0 1 2 3
```

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

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

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

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

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

1.3.2 *Basic vector manipulation*

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

1. Display the contents of the *vector* in the command window by typing its name at the command line. Obviously, this will quickly become useless for very large *vectors*[33].
2. Refer to the Workspace window, – initially, the contents of the vector are displayed (under column Value) and you have to count, but after a certain point, the size (and not contents) of the *vector* is displayed.

   Note that by default, the Size of variables is not one of the displayed columns (instead, it has to be added from Choose Columns right-mouse-button-click menu item)[34].
3. Use the `length` or `size` function (see Box).

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

```
B = B.'
```

will turn the vector `B` into one (assigned back to the same *variable* name) with the same orientation as `C`. [35]

Equivalently:

```
B = transpose(B)
```

(which does exactly the same thing).

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

1.3.3 *Using the colon operator to create vectors*

Refer to the Box on the use of the `colon operator`. But basically, to create a vector of all the integers from 1 to 10:

```
X = [1:10];
```

[33] Try creating a *vector* from 1 to 100,000 and assign it to a *variable*. Refer to the use of the `colon operator` (see earlier).

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

`length`

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

```
length(A)
```

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

`size` (use #1)

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

```
» size(A)
ans =
    1 10
```

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

```
» size(A)
ans =
    10 1
```

(10 rows and 1 column).

[35] Note ... **MATLAB** gives the syntax as `.'`, whereas I always only ever added the `'` bit ... which works ...

`flipud`, `fliplr`

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

```
» B = flipud(A)
```

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

```
» B = fliplr(A)
```

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

1.3.4 *Addressing elements in vectors*

This next bit is maybe the single most important (and weird) part of **MATLAB** (or programming in general). As you go through this section (and also the later one on *matrices*) – it may help to have **Excel** open as a aid to visualize how **MALTAB** represents *arrays* (for the following example, you would enter the 5 numbers, from 0.5 to 2.5, in sequential cells, working down from A1.

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

```
» B(5)
ans =
    2.5000
```

or:

```
» C(3)
ans =
    1.5000
```

(In this text, I will refer to accessing a particular element (or elements) of a *vector* (or *array*) via its *index* as *addressing*. Unless I forget, then I might say something else. You'll have to keep on your toes – don't expect consistency here!)[36]

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

For addressing more than one element of a vector at a time, you can use the `colon operator` (see Box). [37]

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

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

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

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

```
» B = A.'
```

or

```
» B = transpose(A)
```

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

```
» B = A'
```

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

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

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

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

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

To select all elements:

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

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

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

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

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

```
» Y = X.^2
```

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

1.4.1 *Plotting*

First – you will create yourself a dummy dataset to plot. To do this, you are going to need to create yourself a pair of *vectors* – these can have any values in them that you like, but perhaps aim for 1 vector with values counting up from 1 to 10 (or similar) – this will form your *x*-axis, and the 2nd column (the second vector you will create) ... whatever you like. [38]

The command `figure` creates a figure window, which is where **MATLAB** displays its graphical output ... but on its own, without anything in it ... useless. So, lets put something in it, with the simplest possible graphical way of displaying data called `plot`.

With any new **MATLAB** command (*function*), get into the habit of looking up the help text:

```
» help plot
```

Note that you may prefer the formatted on-line version – go to https://www.mathworks.com/help/matlab/ and type help in the search box. (Don't forget that you can also refer to alternative/simplified help provided in the left-hand margin of this text.)

The key information that will get you started appears at the very top of the text that help returns on `plot`:

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

This tells you that you need to 'pass' to `plot`, your *x*-axis data *vector* (by its variable name), followed by your y-axis data *vector* (by its variable name), with the 2 variable names comma separated:

```
» plot(X,Y);
```

(where `X` is the name of your first vector, and `Y` is the name of your second).

plot

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

Its most basic usage is:

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

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

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

Do this. Depending on just what or how random your *y*-axis data was, you should end up with something like Figure 1.1 in a window captioned "`Figure 1`"[39] (although your actual line is likely to look very different from this depending on what vectors you created).

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

Note that by default, **MATLAB** also scales both axes to reasonably closely match the range of values in the two data vectors. In the example here, the default min and max axes limits in fact turn out to be the min and max values in the *x* and y-axis data because the data is composed of relatively simply/whole numbers. If however the maximum *y* value was very slightly larger, you'd see that **MATLAB** would adjust the maximum *y*-axis limit to the next convenient value so as to preserve a relatively simple series of labelled tick marks in the axis scale. In fact, why not try that – replace your maximum data value[40], with a value that is very slightly larger (an example is given in Figure 1.2). [41] 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).

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

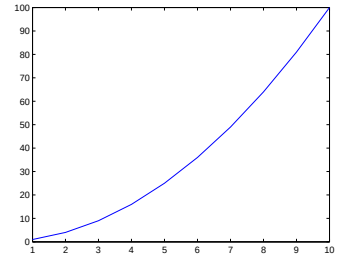

Figure 1.1: Example of the default output of the `plot` function.

[40] Remember that you can replace e.g. the last element of *vector* `Y` with the value `9.9`, by:

```
Y(end) = 9.9;
```

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

1.4.2 *Graph labelling*

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

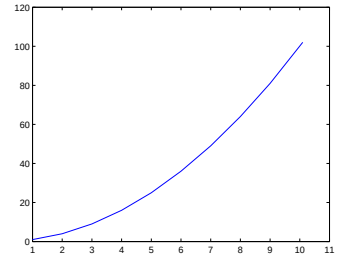

Figure 1.2: A plot illustrating axis auto-scaling (maximum *x* and *y* values now slightly larger than 10 and 100, respectively).

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 (see Box). If you change your mind, simply re-issue the command with the text you would like instead. For example:

```
» xlabel('This is the x-axis.');
» ylabel('This is the y-axis.');
```

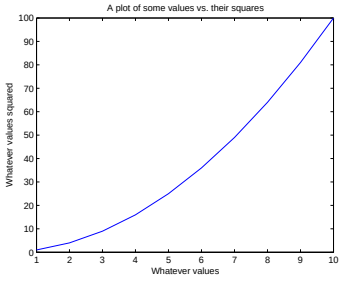

Figure 1.3: A (only very slightly) improved plot.

2. The command for title, perhaps unsurprisingly, is `title` (also see Box). Again, pass the text you would like to appear as a string (in inverted commas `' '`), or pass a the name of variable that contains a string (no `''` is then needed). e.g.:

```
» title('This is a pointless graph the Instructor
made us plot.');
```

3. You might want to specify the axis limits. The command is `axis` (see Box) 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.

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

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

(The notation gets confusing in a narrow box like this – the `...` indicates that the line should be continuous and not broken across 2 different lines.)

Information as to how to use all of these commands can be found via **MATLAB** `help`.

Employing the above 3 suggestions gives rise to the improved plot shown in Figure 1.3, is given in the margin.

Note that in the usage of all the above listed commands, they all require something to be passed within a set of parentheses – `( )`. In fact, they are all **MATLAB** *functions* and require an input (hence the use of the parentheses). Some of the functions require a *string* input, such as the name of the title in `title`, and this must be encased in quotation marks – `' '` to designate it a string rather than a *variable* name.[42] This will all become clearer once you start creating your own *functions* (computer programs) in the next chapter.

[42] You could instead assign a *string* to a *variable*, and then pass the *variable* name (no quotation marks).

`axis`

For once, helpfully, **MATLAB** says: "*`axis([xmin xmax ymin ymax])` sets the limits for the x- and y-axis of the current axes.*" which is about all you need to know (other than the minimum and maximum limits along the *x-axis* are represented by `xmin`, `xmax`, and the minimum and maximum limits along the *y-axis* are `ymin`, `ymax`). Simply stick to the format, with a vector of 4 values (remembering the square bracket notation `[]`), inside of `axis()`, and you should not go wrong!

For example, to scale the plot with the x-axis going from 0-10, and the y-axis from -100-100, you would type:

```
axis([0 10 -100 100]);
```

1.4.3 *Sub-plots*

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

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

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

To place several different plots on the same figure uses the `subplot` command [43]. The `subplot` command is used as: `subplot(m,n,p)` where `m` is the number of rows of plots you want to have in your figure, `n` is the number of columns of plots in your figure, and `p` is the index of the plot you wish to create (see: Figure 1.4).

The basic code then goes something like:

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

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

[43] » help subplot

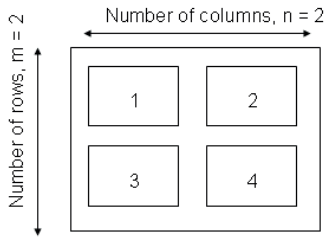

Figure 1.4: Arrangement of subplots.

1.4.4 *Saving graphics and figures*

You might just want to save the figure. (Why create it in the first place in fact if you are just going to throw it away ... ?) Again, you can do this via the *GUI* or at the *command line* [44]. From the *GUI*, you have the option to save the figure in a way that can be loaded later and re-edited – this is the `.fig` format option. Or you can save (export) in a variety of common graphics formats (although once saved in this format, the graphics can only be edited later using a graphics package).

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

(Don't worry for now where you are saving it ... just note the save option, and we'll worry about the computer file system and folder locations later.)

[44] 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`

38 str='do you like bananas?';

1.5 Vectors and arrays #2

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

1.5.1 Creating matrices and arrays

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

1. Enter an explicit list of elements. To enter the elements of a *matrix*, there are only a few basic conventions:
   - Separate the elements of a row with blanks or commas.
   - Use a semicolon, `;` , to indicate the end of each row.
   - Surround the entire list of elements with brackets, `[ ]`.
2. Load matrices from an external data file.
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[45]:

[45] Remember that you can add an `;` to the end of the line to prevent the results of the *variable* assignment being displayed in the Command Window.

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

In math-speak, this would be equivalent to:

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

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

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

of *scripting* (computer programming by another name) that we will see in a few lectures time. Close down this nice toy to leave just the original windows.

Elements in the *matrix* can be addressed using the syntax:

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

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

rows, columns

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

```
» A(1,3)
ans =
    11
```

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

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

Finally – a fundamental way of accessing data that you need to learn and be familiar with, is to employ the *colon operator* to select specific columns (or rows) of data. You'll find that this skill ends up inherent to many of your attempts to process and graph data. For instance, if your (*x*,*y*) data to plot ended up in **MATLAB** workspace in matrix form (it very commonly does) rather than as `2` sperate vectors (as you had when you first plotted anything), you will need to select separately the *x* (e.g. 1st column) data, and the *y* (2nd column) data, and pass these to the `plot` function. For the example of matrix A above, all the first column data can be selected by typing `A(:,1)`[46], which says all the rows (`:`) in the first column. Similarly, all the 2nd column data alone can be selected by `A(:,2)`. (You'll practice this endlessly later on and hopefully get it!)

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

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

1.5.2 Basic matrix manipulation

You can treat *vectors* and *matrices* (or parts of *vectors* and *matrices*), mathematically, as you would treat single values (i.e. *scalars*) but unlike a *scalar*, the transformation is applied to all specified elements of the *array*. This applies for all the basic *arithmetic operators*[47]. For example, for *array* `A`,

```
» 2*A
ans =
```

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

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

```
30 14 22 12
26 2 12 20
42 34 10 6
10 30 40 18
```

Additional matrix operations you will find useful include the `transpose` function (`'`) which flips a (2D) *matrix* around its leading diagonal (columns become rows, and rows, columns)[48]. Also `flipup` (e.g., » `flipup(A)`) and `fliplr` (e.g., » `fliplr(A)`).

[48] 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.

Try out these three operators on your matrix `A`.

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.)

Lastly – earlier you saw how to find the length and size of a *vector* and *array*. Determining the sum of all the elements also comes in handy. For a vector this is straightforward, and

```
» sum(B)
ans = 7.5
```

(assuming that you did not alter any of the values in `B` earlier).

For an *array* (if you check out **MATLAB** help on `sum`) it is a little more complicated. As per detailed in help, "If A is a matrix, then sum(A) returns a row vector containing the sum of each column." If you want the sum of the entire matrix (not just the sums of each column), you have to request:

```
» sum(A,"all")
```

1.5.3 Some matrix math :(

We will not overly concern ourselves with multiplying *vectors* and *matrices* together just yet ... but you should be aware that **MATLAB** can do matrix math. For now, it is worth nothing the difference between `*` and `.*` operators in the context of arrays. For example, consider 2 vectors, `A` and `B`:

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

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

```
» C = A.*B
C =
    1 2 6 8
```

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

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

which you get by typing:

```
» C = A*B'
C =
    17
```

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

An example of the equivalent matrix usage is:

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

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

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

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

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

we would write:

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

1.6 Loading and saving data

There are a number of different ways to load/import data into the **MATLAB** Workspace. Rather than try and tediously list and describe in detail in this section, all the commands and syntax and blah blah blah, we'll take a simple over-view here, and then go through some more detailed and data-based examples as we progress through the course text.

1.6.1 Where am I?

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

MATLAB has a default directory that it starts up in and looks at first. For basic **Windoz** installations[49] of the software, this directory might be:

C:\Users\USERNAME\Documents\MATLAB

(or similar)[50].

You can determine which directory **MATLAB** is currently 'in', either at the command line:

```
> cd
```

Or ... your current directory will be displayed in a toolbar above the Command Window.

It is unlikely that you want to have to save all your files here. More likely, you may have a course folder somewhere, possibly with sub-folders for each week (or whatever), so you will need to change the **MATLAB** directory that you are working in, to match the one where your files are.

Easiest ... is to use the GUI – the toolbar above the Command Window can be used to change the current (working) folder. The contents of the current folder are automatically displayed in the Current Folder pane on the left. (If the Current Folder pane is not present in your **MATLAB** layout, it can be selected form the Layout drop-down menu.) You can also navigate your away around using the Current Folder window by double-clicking on directories (and using the Up One Level icon to move back).

You can also do this (move around directories) at the the command line ... which is much less fun and you are less likely to need to do it this way. The instructions for managing working and file locations at the command line are included in the margin ... for completeness ... you can skip over them if you wish ...[51] ...

[49] At installation, this directory can be specified and hence may not be this one. Also – different operating systems will have different default locations.

[51] Just remember that there are ways of navigating around your computer storage via the command line.

You can change the directory that **MATLAB** is working from by typing:

```
» cd DIRECTORY_PATH
```

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

```
» cd E:
```

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

```
» cd E:\working
```

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

```
» cd('E:\working')
```

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

```
» addpath('E:\working')
```

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

`addpath`
The command `addpath` will add a search path to the **MATLAB** workspace. The syntax is:
`addpath(DIRECTORY_PATH)`
where `DIRECTORY_PATH` is a *string* (characters in between inverted commas) or name of a variable containing a string.

1.6.2 *Loading and importing data*

The simplest way to import data into **MATLAB** is also to use the *GUI* – from the `File` menu, selecting 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 ... The *GUI* can also be used to change the directory you are working from (duplicating the functionality of the `cd` command) and add paths to search (duplicating the functionality of the `addpath` command).

A more flexible way that you can embed in programs, is to use the `load` function (see Box) – we'll concentrate on this in the course.

As a brief exercise and practice in using `load` – download the data file `etheridge_etal_1996.txt` from the course webpage. Note that you need to either save this file directly in the folder you intend to work from (and which you have directed **MATLAB** to by changing its working directory), or copy the file from your computers 'Download' folder if it ends up there, to the working directory.[52]

You might start by viewing the contents of the file by opening it in any text viewer or importing it into **Excel**. This is always a good place to start when importing and processing data in any programming language, as it enables you to see what you are getting yourself in to (i.e. the format of the file, any potential formatting issues, approximate size and complexity of the dataset, etc).

Now, import the data into the **MATLAB** workspace using the `load` command. This looks like:

```
» load('etheridge_etal_1996.txt');
```

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

```
» MYDATA = load('etheridge_etal_1996.txt');
```

is exactly the same as before, but assigns the `load`-ed data to the *variable* `MYDATA`[53] rather than to an automatically-generated one.

Try typing the name of the *variable* that was automatically created (`etheridge_etal_1996`), or the one you chose if you assigned the imported data to a specific variable name, to provide a crude view of the data. A better way to view the contents of the *variable* is through the Variables window – double click on the name of the *variable* in the

`load`

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

```
» load(FILENAME)
```

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

The file might be plain text (ASCII) or a **MATLAB** workspace file (see below). To force **MATLAB** to treat the file input as ASCII or a **MATLAB** workspace file, pass a second parameter (separated from the filename by a comma) – `'-ascii'` for ascii, and `'-mat'` for a **MATLAB** workspace file, i.e.

```
»load(FILENAME,'-ascii')
```

would specify that a plain text (ASCII) file is loaded.

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

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

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

(all one line)

Newer versions of **MATLAB** can tell between ASCII and **MATLAB** workspace formats, and the `'-ascii'` bit is generally not necessary.

[52] In eLearn – click on the file to take you to a page where the contents are displayed.

At the top of this page, you should see: Download etheridge_etal_1996.txt (5.27 KB) with the filename highlighted. Right-click over the link, and choose Save link as ... make sure you are saving as a .txt text file and not as a 'web page'.

[53] Remember that all capital letters words indicate a place-holder for your own, here: variable name.

MATLAB Workspace window. This should open up a spreadsheet-like window in which the data can be viewed, sorted, and even edited.

For practice, try plotting the data[54] and remembering to label the figure appropriately[55]. However ... remember, the format of the **MATLAB** `plot` function is:

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

so you will need to specify each column of the data (i.e. each *vector*) separately and explicitly because the data you loaded is in the form of a single variable containing a 2D array of values.[56]

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

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

Or, if you are comfortable with more complex, single lines of code, go straight for the kill:

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

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

(If you plot ends up looking like Figure 1.5 – re-read the instructions carefully.)

[54] using `plot`

[55] FYI: the first column of the data and x-axis is year, and the 2nd column of the data and y-axis is the mixing ratio of CO_2 in air in units of ppm.

[56] If you just type `plot` and pass the (here: default) name of the data array:

```
»plot(etheridge_etal_1996);
```

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

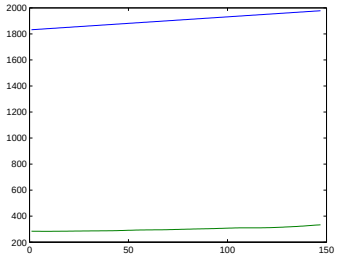

Figure 1.5: Result of simply throwing the entire data matrix at `plot` ....

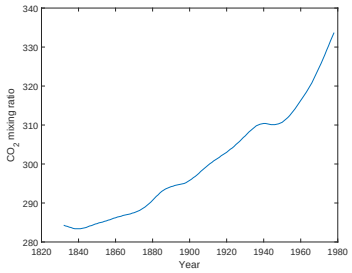

Figure 1.6: Spline fit to measured changes in CO2 concentration in Law Done ice core, following *Etheridge et al.* [1996].

1.6.3 *Saving and exporting data*

For your information – specific variables can be saved in a plain text (ASCII format) by means of the `save` function (and then re-loaded in using `load`). You have to specify that you want a text format (rather than the default **MATLAB** .mat workspace format) – see Box.

1.6.4 *Loading and saving the workspace*

The entire workspace (including all variables and their values, or just the values in a single variable if you wish) can be saved to a file and then later re-opened. The file format is specific to the **MATLAB** program and the file-name extension by default is `.mat`[57] You might find this very helpful to use in long lab exercise or large modelling projects, particularly if you do not come back to work at the exact same computer each time or wish to use continue the same piece of work on a laptop elsewhere. Try saving the current Workspace, then close down the **MATLAB** program. Re-running it, and then loading in your saved `.mat` file. [58]

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

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

[58] This sequence is going to look something like:

```
» save MYSTUFF
» exit
```

...

```
load MYSTUFF
```

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

`save`

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

```
» save(FILENAME)
```

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

In contrast:

```
»save(FILENAME,A,'-ascii')
```

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

For example, if you had a variable `bananas`, and you want to save the contents (data) as the file `bananas_data.txt`, you would write:

```
»save('bananas_data.txt',
...'bananas','-ascii')
```

(remembering this should be a single line and the `...` simply indicates that the line should be continuous)

1.7 Basic data processing (and yet more plotting)

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

1.7.1 Sorting data (in arrays)

As an example to kick-off some data-processing tricks, load in the dataset of ('proxy') reconstructed atmospheric CO_2 concentrations spanning the Phanerozoic: paleo_CO2_data.txt. You can just import it into **MATLAB** using the `load` function as before – remember the specific syntax of load:

```
» load(FILENAME)
```

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

```
» load('paleo_CO2_data.txt')
```

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

If you view the contents of the variable `paleo_CO2_data` (or whatever you might have passed the `load`-ed data to), you will see that there is a slight complication – unlike the ice core CO2 dataset, you now have 4 columns in this array[59]. The first column is age (Ma), the second the mean CO_2 value, while the 3d and 4th columns are the low and high, respectively, uncertainty limits of the estimated past CO_2 value.

Recalling how to reference specific columns of data in a matrix[60], and either referencing the columns of the array directly, or creating yourself separate vectors `X` and `Y` (see earlier) – plot the mean paleo CO_2 value as a function of age (in Ma). If you closed the previous Figure window (see earlier), it is not essential to explicitly open one (using the `Figure` command) – when you use the `plot` command, if there is no open Figure window, **MATLAB** will kindly open one for you. How thoughtful. The result of:

```
» plot(paleo_CO2_data(:,1),paleo_CO2_data(:,2));
```

(plotting the 2nd column out of the 4 columns, vs. the 1st column), should be something like 1.7 ...

O dear ...

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

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

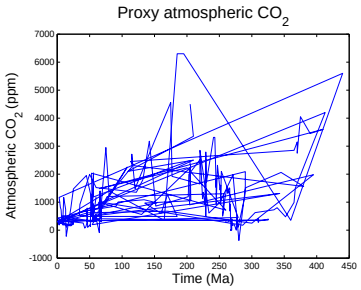

Figure 1.7: proxy reconstructed past variability in atmospheric CO2.

So ... that was not so successful. Why? What is happening in the default behaviour of `plot`, is that the (x,y) location defined by each subsequent row of data is being joined to the previous one with a line. This was fine for the ice-core CO2 example dataset because time progressed monotonically in the first column, e.g. the data was ordered as a function of time. If you view the paleo CO_2 data, this is not the case and time (age in Myr) does not progress monotonically in an always-getting-older (or always-getting younger) fashion.[61]

[61] 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:

1. You could import the data into **Excel**, then re-order (sort) it, then export (save) it, then re-load it ...
2. You could sort it in **MATLAB** using the *GUI* variable view window. But lets not cheat for now.
3. You could sort it in **MATLAB** at the command line. How? Well, a reasonable gamble is to try:

   ```
   » help sort
   ```

   However, reading the help text carefully (and you can always try it out and see what exactly it does if you are not sure), you will find that the *function* `sort` will sort all columns independently of each other, whereas we want the first column sorted and the remaining columns linked to this order. So this is not the *function* that you are looking for.

`sortrows`

In its simplest usage:

```
» B = sortrows(A)
```

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

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

Beyond this usage, if you wanted to sort by a different column to the first, you would use:

```
» B = sortrows(A,N)
```

where `N` is the column number.

You can also sort in descending order:

```
» B = ...
sortrows(A,'descend')
```

(ascend is the default).

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

```
» sorted_data = sortrows(paleo_CO2_data);
```

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

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

it looks rather better – Figure 1.8. (This is a good illustration of a guess of a *function* that was not quite what was needed, but following up on the `help` suggestions leads to a more appropriate *function*.) At least now the curve is reminiscent of past changes in global temperature and the geological Wilson cycle, with high CO_2 values in the Cretaceous and Jurassic and then lower again in the Carboniferous (roughly matching the progression of ice and hot house (and then back to recent ice ages) climates).

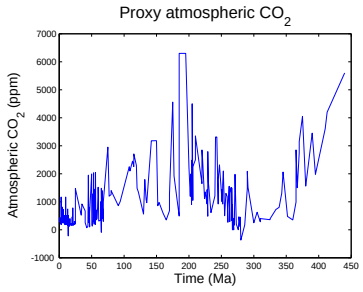

Figure 1.8: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

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

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

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

1.7.2 *Data scaling*

As an example practicing some basic data scaling: download the historical global temperature anomaly dataset[62]:

temperature_globalanom.txt

[62] NOAA

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

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

Plot the annual mean temperature anomaly for the full range of years, as per Figure 1.9. (plus labels, title, etc etc), remembering again that you cannot pass only the entire array to plot, as per:

```
» plot(data1);
```

but instead, must pass the 2 vectors (of x, and y-axis data), separately, i.e.:

```
» plot(data1(:,1),data1(:,2));
```

(where e.g. `data1(:,1)` specifies all the rows of the 1st column of array `data1`).

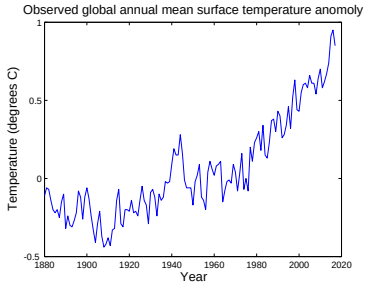

Figure 1.9: Observed annual global mean surface temperature anomaly (compared to year 1910 to 2000 average).

We are now going to transform the data so that it is as an absolute, rather than relative, temperature. The 20th century average global temperature across land and ocean surface is apparently 13.9°C. So first – change the temperature anomaly data into absolute temperatures – you'll do this by by adding the 20th century global average value, 13.9, to all the data values in the second column of your array.[63] (If you are being good and reading the margin text ... the answer is there ...)

Now re-plot.

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

```
» C = A + B;
```

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

```
» A = A + B;
```

In your specific example:

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

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

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

$$T_{(°F)} = 1.8 \times T_{(°C)} + 32$$

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

For this, you will need to take your data (which is the 2nd column of the array), e.g. `data1(:,2)`, multilpy it by 1.8 as per the equation (`1.8*data1(:,2)`) and then add 32 to it (i.e. `1.8*data1(:,2) + 32`)[64]. And ... assign the results of this to a new vector or array. Or you can replace the original column (if you are feeling totally confident), e.g.

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

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

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

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

[64] If you have any doubts as to the order in which the operators are applied, add a set of parentheses, e.g.
`(1.8*data1(:,2)) + 32`

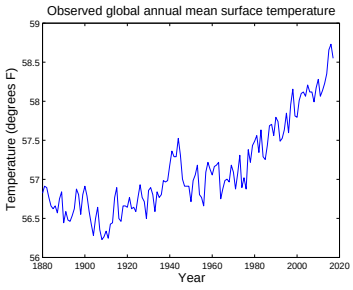

Figure 1.10: Observed annual global mean surface temperature.

1.8 Nicer graphing

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

1.8.1 Modifying lines/symbols in `plot`

The first deviant activity you can engage in with `plot`, is to graph the data without the line joining the points. Scrolling a little the way down » `help plot`, it turns out that there are a number of options for color, line style, and marker symbol that you list together as a single parameter, straight after the parameters for x and y vectors. By default, **MATLAB** plots a solid line in blue with no marker points. Obviously, we could forego the sorting and plot a sane graphic (hopefully) by plotting just points and having no line between them. Hell, you could even be radical and use a different color ... Or, you could specify a symbol and no line. The choice of colors is your oyster, as they (almost don't) say. e.g. Figure 1.11.

A summary of a few of the more common plotting options is provided in the Box. For any previous plot you have made, try changing the line style (or no line), and the marker shape and color.

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

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

you now add, following a comma, the point and line style option you want, which must be encased in a pair of inverted commas, e.g. for a red, dashed line, you would type:

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

and for blue circles (with no line):

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

1.8.2 Plotting multiple data-sets

So far, so good. But so boring, although simple marker-only and joined-by-line plots have their place. For a start, the original data-set included an estimate of the uncertainty in the CO_2 reconstructions in the form of the min and max plausible value for each 'central' (best guess?) estimate. As a visual indication of the uncertainty in the CO_2 reconstructions, one could plot the min and max values as points, using different symbols. This requires a further little trick in **MATLAB**, which involves the command `hold`. This is nice and simple and takes the additional (2nd) word: `on`, or `off`.

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

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

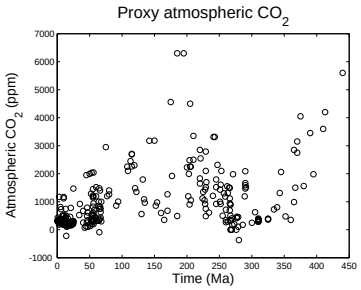

Figure 1.11: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

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 the different lines or

`hold`
According to **MATLAB** help:
`hold on` – retains plots in the current axes so that new plots added to the axes do not delete existing plots.
`hold off` sets the hold state to off so that new plots added to the axes clear existing plots and reset all axes properties.

sets of points are, you can add a legend with the `legend` command. For instance, if you have managed to successfully plot the mean CO_2 values as discrete black circles, and the minimum and maximum uncertainty limits as blue and red lines, respectively, you could type:

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

(all one line) and it should end up looking like Figure 1.12.

It is not something to worry about here (or even necessarily try), but just to make you aware that **MATLAB** has a function – `errorbar` – to help you to visualize errors, including non-symmetric errors, relatively easily. (See **MATLAB** help and the margin Box.)

The complication here is that none of the options for `errorbar` allow for absolute values to be used for error plotting. So to use it in this particular example, we need to replace the absolute min, max column values, with their respective deviations from the mean value (2rd column). Hopefully, you can see the way to do this. For instance, to create an error for the maximum estimate (4th column):

```
sorted_data(:,4) = sorted_data(:,4)-sorted_data(:,2);
```

where we are saying: take all the values in the 4th column of the array, subtract the values in the 2nd column of the array, and assign the values back into the 4th column (hence replacing the absolute maximum estimate with the deviation of the maximum from the mean, i.e. the +ve error). (And then do similarly for the minimum values of the 3rd column.)

1.8.3 *Changing label font size (and type)*

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

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

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

```
'FontSize',14
```

`legend`

According to **MATLAB** help:

The command `legend` on its own, by default will label your datasets, 'data1', 'data2', etc etc, which is probably not what you want ...

So you have to supply strings – one for each dataset plotted:

```
legend('DATANAME1','DATANAME2',
...,'DATANAMEN');
```

So if your plot had 2 greenhouse datasets – 'CO2' and 'CH4', you would type:

```
legend('CO2','CH4');
```

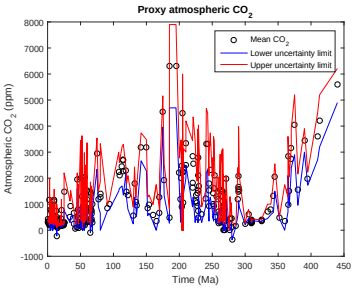

Figure 1.12: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

`errorbar`

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

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

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

The first additional parameter specifies the aspect of the axis label that you wish to change (here: `'FontSize'`), and the 2nd parameter of the pair, is the (new) value (here: `14`).[65] Similarly, the y-axis label and title text size can be adjusted in exactly the same way.

Other property parameters[66] that might be useful (to change), are (with example changes):

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

(You can, of course, also adjust everyting about the look and feel of your plot via the figure window *GUI*.)

1.8.4 *Scatter plots*

Returning back to the Phanerozoic proxy (CO_2) data, we can now put a different (graphical) spin on it.

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

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

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

Perhaps a little disappointingly, the default (Figure 1.13) (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 1.14.

[65] See help on `xlabel`.

[66] Again – refer to **MATLAB** help.

`scatter`

"*... creates a scatter plot with circles at the locations specified by the vectors x and y. This type of graph is also known as a bubble plot.*"

The simplest usage is:

```
scatter(X,Y)
```

where is works pretty much like `plot`, except without the joining line segments. (`X` and `Y` are *vectors* of the same length)

```
scatter(X,Y,20)
```

specifies the marker size (in points), and:

```
scatter(X,Y,20,'filled')
```

then fills the markers. Finally:

```
scatter(X,Y,20,Z,'filled')
```

colors the points according to the data in the *vector*, `Z`, or instead scales the marker size with `Z`, via:

```
scatter(X,Y,Z,'filled')
```

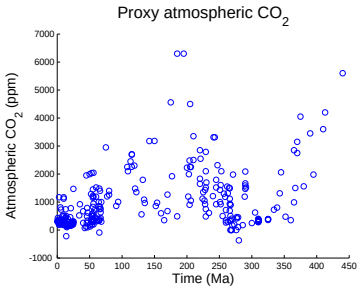

Figure 1.13: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

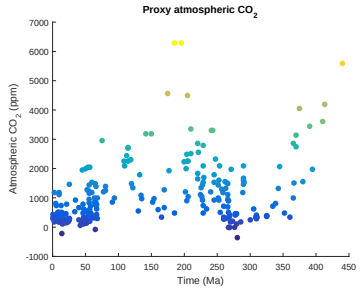

Figure 1.14: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

1.8.5 *Simple 2D data and bitmap visualization*

There are 2 different simple **MATLAB** commands for visualizing a 2D dataset (i.e. a matrix) as a bitmap image (and via a 3rd command, viewing various bitmap photo and image format files too).

As something (2D data) to play with – `load` in the data matrix: `model_grid.txt` (labelled as: '2D model grid data'). Then, view the data in the array viewer, just to get a feel for what you are dealing with here (although you are unlikely to be much wiser after doing so).

Now, employ the `pcolor` function in its simplest possible usage (see Box) to visualize the data – i.e. you simply pass the name of the variable containing the data matrix into the function. You can see (Figure 1.15) that it is ... something. Maybe a little like the continents, but up-side-down at the very least. What to do?

Well, it is a good job that you remember how to re-orientate arrays, right? If you guess right first time (three different basic transformations of a matrix were described), you will get Figure 1.16.

Next try something very similar, but use the `image` function – refer to the help box but note that for viewing a data matrix its usage is the same as that for `pcolor` (you pass the variable name containing the 2D data).[67]

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

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

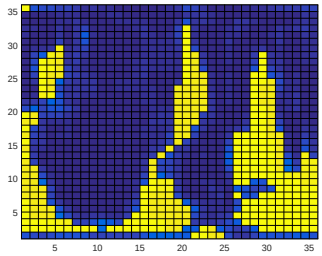

Figure 1.15: A 2D plot of some random gridded model data.

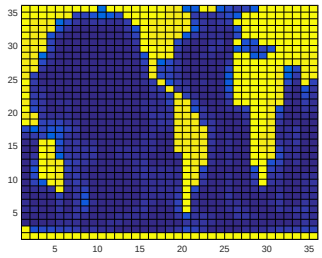

Figure 1.16: A 2D plot of some random gridded model data ... but with the underlying data matrix re-orientated before plotting.

[67] Now the model grid is the correct way around! I have absolutely no idea why and why it is reading the matrix dimensions differently from `pcolor`. I am sure you could **Google** and find out. But you would have to actually care first.

`pcolor`

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

```
pcolor(model_grid);
```

`image`

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

1.9 *Further matrix math (systems of equations)*

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

As an example – consider the Great Lakes – the largest lake system in the world. They have on their shores some of the greatest cities ... as well as some of North 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 1.17, together with the mean volumes of the lakes and the annual flow rate of water out of them.

We can analyse the net result of a cocktail of heavy metals pouring into each lake, the amount dependent largely upon the population within the catchments of the lake. The assumed input rates to each of the 5 lakes are given below.

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

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

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 ...)

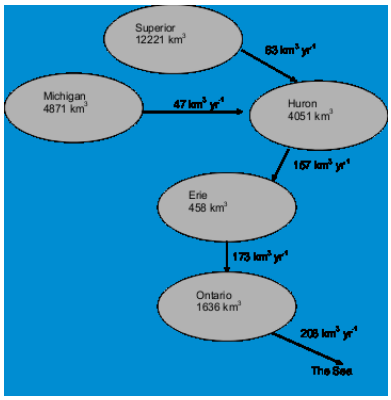

Figure 1.17: Lake volumes and river flow rates in the Great Lakes system.

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 this 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.[68] 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.[69]

[68] 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.

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

2
Elements of ... programming

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

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

2.1 *Introduction to scripting (programming!) in MATLAB*

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

Your wish is granted! In **MATLAB** (and all programming languages), it is possible to store all of your commands in a single text file, and then request that they (i.e. the list of commands) are all executed (sequentially) at one go. **MATLAB** gives this text file a fancy name (because it is a very fancy piece of software, after all) – a *script*[2], otherwise known as an *m-file*. To create a new *m-file*; from the File menu, select Script (a common type of *m-file*)[3]. You will see a text editor (more fancy-ness) appear in front of your very eyes, containing your requested (but currently empty) *m-file*. Save the *m-file* to your working/course directory of choice. (Alternatively, simply create a new (blank) text file and save it with the extension .m, rather than e.g. .txt – this creates you a (script) *m-file*, illustrating that an *m-file* is nothing more than a text file with a .m file extension.)

From an *m-file*, you can issue all the **MATLAB** commands you previously would have entered individually, line-by-tedious-line, all at once. Furthermore, having created and saved a **MATLAB** script, it can be executed again and as many times as you like.

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

OK – you are now ready for your very first program ... inevitably ... this has to be to print `'Hello, World'` to the screen. No, really. (**Google** it.)

Create a new *m-file*, calling it e.g. hello_world.m (remembering that spaces are NOT allowed in filenames). Make sure that it is saved in the same directory that **MATLAB** is working from ...

You are going to use the function `disp` (see margin help box and/or type `» help disp` to find out the **MATLAB** *function* syn-

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

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

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

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

```
disp(STRING)
```

where `STRING` is a .... *string*, you get the *string* displayed as text at the command line.
In general, you can pass the name of any variable

```
disp(VARIABLE)
```

and get the contents of `VARIABLE` displayed.
Note that the difference between using `disp` and simply typing the variable name:

```
» VARIABLE
```

is ... well, find out for yourself!
Note that there is no effect of including the semi-colon (`;`) at the end of `disp()`.

tax and usage). This command (*/function*) will print to the screen, either any text you specify (in inverted commas), or the contents of any *variable* (you pass the variable name to `disp` (without inverted commas)). For now, simply pass the text directly.

Your program needs just a single line in the *m-file* (not at the command line):

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

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

```
» hello_world
```

Your first program is a success!(?)[5]

You could extend this to a mighty 2-line program by defining the string as a variable on the first line of the *m-file* (i.e. the 'program'), and displaying the contents of the variable (on the second). In a new *script m-file* (saved as e.g. hello_world2.m), add the lines:

```
message = 'hello, world ...  again!';
disp(message)
```

and then save and run it.[6,7]

For further practice – pick one of any of the previous exercises in which multiple lines of code were required, such as loading and then plotting a data set – place these lines of code into a new *m-file* (either by re-typing them in or copying them out of the Command History window), save the file, and then run it my typing its name at the command line (omitting the .m extension).

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

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

[7] If you get the message repeated twice, you might have committed the semicolon form the end of the first line.

2.1.1 *Programming good practice*

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

- Choose helpful *variable* names so that it is clear what each *variable* represents. Avoid *excessively* short names, except for simple index and counting *variables*. At the other extreme – excessively long names, which the might be wonderfully descriptive, can lead to even simple calculations stretching over multiple lines of code (which can make it more difficult to see what is going on in the code overall). It is also very easy to create typos in trying to use very long *variable* 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*[8], and is ignored by **MATLAB**.

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

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

```
close all;
```

This command close all currently open figures, plots, images, etc. so that if you repeatedly run the script such as you might in developing and debugging it, you don't end up with 1000000000s of Figure windows open ...

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.

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

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

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

This example also illustrates one possibility for a consistent *variable* naming convention – constants (*variables* which never change in value) start with a `const_` and parameters (variables whose values might be changed) with `par_`, temporary ('local') variables with `loc_` and variables passed into and out of the function: `dum_`. Note the use of the semi-colon at the end of every line to prevent (here unwanted) printing of results to the screen. (Don't worry about what a *function* is yet ... just not the degree of commenting and that there is some sort of consistent and meaningful naming convention.)

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

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

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

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

If it (a line) starts with a percentage symbol, then **MATLAB** ignores it and you can type whatever you like after it (on the same line).
Also note, if it helps – you can run a single line of code over 2 lines of the file by adding:

```
...
```

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

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

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

The book schematic structure of this program (*script*) is shown in Figure 2.2.[9]

Finally, and related to the next subsection – code in stages, testing the (partial) code at each step. Do not try and write all the code in one go and only try it out at the end[10].

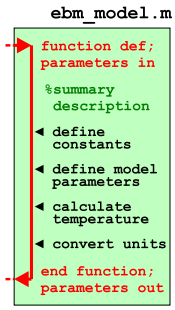

Figure 2.1: Schematic of the example program.

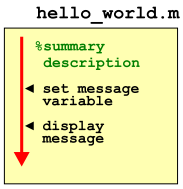

Figure 2.2: Schematic of the Hello World program.

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

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

2.1.2 *Debugging the bugs in buggy code*

What programming is mostly about is not writing new code so much as debugging[11] what you have already written. Key then, is to reduce the incidence of bugs occurring in the first place, and when they do occur, firstly to have code that lends itself to debugging and secondly, knowing how to go about the debugging. The first two facets are at least partly addressed through good programming practice (see earlier)[12].

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

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

Here is an example to try out to start to see what might be involved in debugging, loosely based on a previous plotting example – go create a new *m-file* called: plot_some_dull_stuff.m[13]. Then add the following lines to the file – you can copy-paste from the PDF file, but you will have to change the ' characters (which differ between those used in the PDF and what MATLAB wants):

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

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

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

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

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

Actually ... if this were your program, you should have paid attention to earlier and not have written it all at once before testing it! But at least **MATLAB** is giving you some sort of feedback. The actual error reported might not always mean that much to you but the line number at which the problem occurred is gold-dust. The line of code is does not like is line 6[14], which is:

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

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

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

```
» cos[0.0]
```

to which you get told:

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

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

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

Other strategies for helping debug include:

1. Checking what the values of the variables were at the point at which the program derp-ed (broke down) – the current (and the point of program crash) variable values are listed in the Workspace window.
2. Changing the relevant variable value(s) (here `x`) and re-typing the problem line to see if it makes a difference[15].
3. Commenting out (%) lines of code temporarily, or adding in additional (temporary) lines of code, and re-running. Where coding in bite-sized chunks is an advantage in this respect, is that if a program stops working after you have added a new section of code, you can go comment out the new code (never normally just delete it all), check that the original section of code still works, and then line-by-line, un-comment the new code until the problem line is found.
4. You can also put your program on hold just before the problem line and explore the state of the variables at that point (see Box), although in this particular example of a bug, **MATLAB** does not allow this, presumably because if feels that the mistake is simple and can be easily fixed.

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

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

```
Undefined function or variable 'k'.

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

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

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

Debugging – breakpoints

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

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

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

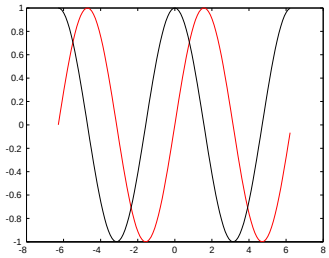

Figure 2.3: Output from the (bug-fixed version of) plot_some_dull_stuff *m-file*.

2.2 *Functions*

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

A *function* is defined (and differentiated from a *script*) by a special line at the very start[16] of the *m-file* (see Box). You must follow the **MATLAB** syntax exactly in defining a function.

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

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

You can't, of course, go re-defining pre-defined **MATLAB** function names[17]. So how about if in your work, you found you frequently needed to use the square of the sine of a number. You could keep writing:

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

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

In this example, the contents of your *m-file*, which here we'll call sin2[18], would look like:

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

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

Try this out – create a new blank *script m-file*, add this code (and add comments!!!), save the file, and run it.

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

```
» sin2(0.5)
```

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

```
» Y = sin2(0.5);
```

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

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

Functions

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

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

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

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

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

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

```
» calc_stuff()
```

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

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

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

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

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

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

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

```
end
```

[17] Actually you can, but it is best not to.
[18] And hence filename sin2.m.

 str='do you like bananas?';

NOTE that you can make a new function in 2 different ways:

1. Create a new (blank) *script m-file*.
   By adding the function definition line on the very first line of the file (e.g. `function [] = calc_stuff()`), and `end` at the end, you turn the *script* into a *function*.
2. From the **MATLAB** Editor or main window menu, you can also chose: New -> Function.
   This creates you a generic *function* template file with a bunch of lines of stuff already in it:

```
function [outputArg1,outputArg2] = untitled3(inputArg1,inputArg2)
%UNTITLED3 Summary of this function goes here
% Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

   If you find this too confusing, particularly early on in the course, just stick to method (1).

NOTE2: You do not want **MATLAB Live Script**.

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

When you are happy with this, create one with 2 inputs (refer to **MATLAB** `help` on `function` and/or refer to the previous Box), that returns a value equal to the sine of the first input, divided by the cosine of the second input[19], i.e.

$y = \frac{sin(x_1)}{cos(x_2)})$

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

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

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

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

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

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

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

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

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

 str='do you like bananas?';

2.3 *Conditionals '101'*

2.3.1 `if ...`

One of the most important programming constructs is the *conditional statement*, in which whether one or more *statement*(s) are executed (and hence the overall outcome) is conditional on the 'truth' or otherwise (i.e. it being true or false) of a given *expression*.[20]

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

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

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

   ```
   if happy
   ...
   ```

   where `happy` is a variable which is `true` (`1`) or `false` (`0`).

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. The result of a *relational operator* (see earlier), i.e. one of e.g.:

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

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

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

   where `A` and `B` are numbers.

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

A rather computer programming textbook-like example now follows ... :(

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

But before we worry about anything else (e.g. how to apply a *conditional* statement), you'll need to know about inputting information into a **MATLAB** program from the keyboard[21]. Amazingly, you can guess (I actually just did) the command for requesting input – it is `input` (for 'input' – a rare occasion when everything is logical and simple!) (see Box).

[20] Pause ... and deep breath.

Conditional Statements

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

The basic `if` structure is:

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

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

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

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

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

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

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

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

Armed with this important new information (how to get **MATLAB** to ask for input and then receive and do something with keyboard input) – firstly create a blank *m-file* and save with a 'suitable' filename. (You are going to be typing code into the *script m-file* (not at the command line.) Maybe add a header line *comment* (a 1st line or lines starting with a %) to remind you what this *script* is going to do.

Secondly, (and on the next line after the comment line) – define the text (question) that you are going to ask and assign this string to the variable `MY_QUESTION` (substitute your own variable name here).

Then place the `input` command (on the next, now 3rd line) for string input, and assign the input string to the variable `MY_ANSWER` (again, you can pick your own variable name).

You should now have a program consisting of 3 (or more, depending on how many *comment* lines you include) lines – an initial *comment* line, a line defining the question and assigning this string to a handy variable (`MY_QUESTION`), and a line taking the results of the `input` function, and assigning it to a second variable (`MY_ANSWER`). The structure of your program should look like Figure 2.4. To help you out, a complete program looks like:

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

Run the program thus far. You should see the question displayed, and when you type in an answer and hit **RETURN**, the program will end. Because your *m-file* is configured as a *script* and not a *function* (see earlier), you can see the variable `MY_ANSWER` in the variable list and you can hence check its value – it should contain a *string* with the answer you gave to the question. Make sure it all works like this so far.[22]

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

`input`

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

For inputting a numerical value:

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

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

For inputting a string:

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

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

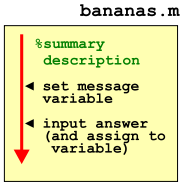

Figure 2.4: Schematic structure of the simple bananas question program.

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

Copy your previous *m-file* and give it a new name, so that you retain a copy of your previous working program before you start developing it further. Work from the new file.

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

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

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

where `[RESPONSE]` you will later replace by a message that you will display using the `disp` command that you saw before. (In this stupid example it might be: 'Correct – they are a great fruit!'). In this (now illegal!) usage, we are trying to ask whether the contents of the *variable* `my_answer`, are equivalent (the `==`) to the string `'yes'`.

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

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

Or, we could have made this more compact:

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

Your code should now comprise something like the 3-5 or so lines from before (comment, define question, get input) followed by 4 lines of code of the conditional structure, comprising: the `strcmp` function, the `if ...`, use of `disp` to display a message, and lastly, `end`. The structure should look like Figure 2.5[23] or if you assign the message to a 2nd variable, like Figure 2.6. A complete example program ... to help you follow all the above, would look like[24]:

`strcmp`

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

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

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

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

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

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

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

(Please – do not just copy-paste the code ... write your own version of the code and only use this code as a guide.)

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

Note (but not necessary try out) that you can also turn this around, and test for an answer that is not 'no' (the ∼ is making the test, not 'no'), i.e.

```
if ∼strcmp(MY_ANSWER, 'no')
  [RESPONSE]
end
```

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

Next, you might display an alternative message is the answer is not 'yes'. Refer to **help** / the margin Box on `if ...` and note that you can extend the structure with an `else` which would be followed by a line displaying the alternative message (e.g. 'Then you need to get a life, apple-lover.')[25].

Copy your previous *m-file* and give it a new name, so you retain a copy of your previous working program before you start developing it further. Work from this new file.

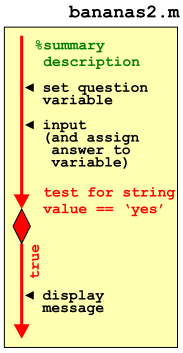

Figure 2.5: Schematic structure of the extended bananas question program.

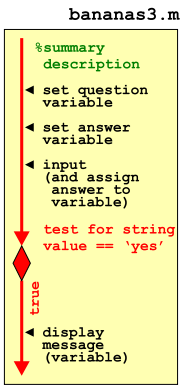

Figure 2.6: A slight variant on the schematic structure of the extended bananas question program.

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

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

```
if strcmp(MY_ANSWER, 'yes')
  [RESPONSE1]
else
  [RESPONSE2]
end
```

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

```
if strcmp(MY_ANSWER, 'yes')
  [RESPONSE1]
elseif strcmp(MY_ANSWER, 'no')
  [RESPONSE2]
else
  [RESPONSE3]
end
```

Here – we are now adding an `elseif ...` line (followed by its specific message) (and see Box/**help**).

Try this extension to your program and test it fully – inputting a 'yes', a 'no', and some other answer, and confirming that you get the correct message displayed.

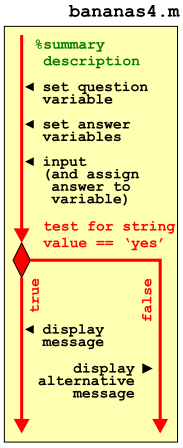

Figure 2.7: Schematic of the bananas program using the `if ... else ...` construct (and displaying alternative messages).

Continuing to beat this same tired example to death ... what if some wise-crack answered 'YES' rather than 'yes'?[26] One could write:

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

This will work, but you might note that you have had to exactly duplicate the RESPONSE line. If instead of displaying a simple message, a complex calculation was carried out – all the lines of the code following the `if ...` would have to be exactly duplicated after the `elseif ...` . While it might seem trivial to simply copy-paste the required lines, this is[27] dangerous – if the first set of lines are ever changed (due to a bug-fix or simple further development of the code), the same changes MUST then be exactly duplicated in each and every instance, or the code will not longer work correctly. This is *very* easy to forget to do, particularly for extensive code or

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

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

code that you have not looked at for ... years. Code duplication also makes the overall code unnecessarily long (and hence harder to look through).

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

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

In code, this is written:

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

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

So – go modify your code to allow for a 'YES' or a 'yes'. Hell, try allowing for a 'Y' or a 'y' as well.[28] (You could extend it to 'no' also but I think you get the point ...) Be careful with all the nested parentheses – an source of mistakes/bugs. You might write it like this, for example:

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

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

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

How many bananas could you eat in a day? I bet it is less than ten. We'll let the computer ask and if the answer is 10 or more, you (the computer) replies: 'liar!'.[29]

The basic code is very similar to before. Create a new *m-file*, add a comment line, define your question string ('How many bananas do you think you could you eat in a single day?') and then get **MATLAB** to ask it and pass back whatever is entered in at the command line.

The only difference at this point – refer to the usage of `input` (see earlier Box) – is that we want a number input rather than a string. So instead of:

```
MY_ANSWER = input(MY_QUESTION,'s');
```

where **MATLAB** is going to treat whatever you enter as a *string*, we have:

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

where **MATLAB** is going to treat whatever you enter as a *number*.

In the if statement, we now want to test whether the value of `N_BANANAS` (which you replace with your own variable name) is greater or equal to 10 (or equivalently, greater than 9), i.e.

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

 str='do you like bananas?';

```
if (N_BANANAS >= 10)
   [RESPONSE1]
else
   [RESPONSE2]
end
```

or equivalently:

```
if (N_BANANAS > 9)
   [RESPONSE1]
else
   [RESPONSE2]
end
```

Write some code along these lines (using this structure) and get it working. Start with a new blank *script m-file*. Your code will look almost identical to previously, except for the different usage of `input`, and the form of the conditional.

Feel free to switch fruit / fruit consumption threshold, question/answers, or whatever.

2.3.2 `switch ...`

A slightly less commonly used alternative to `if ...` is:

`switch ... case ...`

and is helpful in the case of multiple possible correct answers and/or multiple different answers.

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

Using `switch ... case ...` the code might instead look like:

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

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

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

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

```
case 'banana'
```

for a string, and for a number:

```
case 10
```

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

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

Conditional Statements (2)

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

The basic `switch` structure is:

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

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

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

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

You can also have multiple case possibilities:

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

 str='do you like bananas?';

2.4 *Loops '101'*

The next main program construct that you are going to see is the *loop*. There are a number of different forms of this in **MATLAB** (see *loops* Box) (and also in other programming languages), but the basic premise is the same – a designated block of code (one of more lines of code[30]), is repeated, until some condition is met. That condition might be something as simple as a count having been reached, e.g. the block of code is always executed *n* times, or the condition might be slightly more complex and involve a *conditional statement* (see later). Will explore a very basic loop though an example, almost as contrived as for conditionals :o)

2.4.1 `for ...`

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

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

Consider, or rather: **create a new *script* m-file**, and add following code for a simple loop:

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

Save the file and then run the *script*. What did it do?

I bet you have absolutely no idea! It actually cycled around ten times, counting from `n=1` through `n=10`, but you would not know it as there was no code within the loop to do anything and tell you anything about it.[32]

There are 2 alternative (but very crude debugging strategies) you could take in order to investigate the behaviour of the code[33]:

1. In the **MATLAB** Editor, add a *breakpoint* to the `end` line of the loop. When you run the script, the program stops after the first time around the loop (and as it reaches the `end` statement). You can now e.g. interrogate the value of `n` by hovering your mouse pointer over it. You can also Continue (button in the icon bar) the

***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 (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`.

[30] 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.

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

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

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

program, and it will then go around the loop once more and then stop again at end.
Typically, there will be one or more lines of code within the loop, and you could create the *breakpoint* at the first line of code (within the loop).

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

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

and it will spit out the value of `n` each time around the loop.
Or, you could print the value of `n` 'properly'[34], e.g.

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

You could 'tart this up'[35] further by creating a string that provides more explicit information back to you, which is when you really need to use `num2str`, e.g.

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

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

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

(but they work the same – check it).

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

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

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

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

```
disp(n)
```

[35] (make look nicer)

Loops in action. So, consider the following (somewhat contrived) problem – you want to be able to enter a series of numbers and return their sum (although equally one could perform and return all sorts of statistics).[36] The basic code is simple and you can try it out by **first creating a new (*script*) m-file**.

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

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

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

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

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

then please do! There is no particular computational penalty in **MATLAB** for creating as many variables as you like and breaking down code into multiple lines.

So far so good. But what if you wanted 4 numbers summed (don't write all this out!) ...

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

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

numbers that you might need to input and process? An impossible and thankless task ...

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

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

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

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

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

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

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

and the *loop* with go around 4 times as the counter `n` counts from 1 to 4 (`MAX_N`) in increments of 1 (the default behavior of the *colon operator*). Each time around the *loop* the block of (2 lines of) code is executed and a number is inputted. But what is still missing? Try this (in a new m-file):

```
for n = 1:4
   my_question = 'Please enter a number:  ';
   x = input(my_question);
end
```

and see if you can see what is going on, or rather, not going on. If you think that it is not working as expected – try some debugging (i.e. adding one (or more) `disp` statements within the loop code, or add a *breakpoint* within the *loop*). (You should have deduced that what is not going on, is that you are not learning what the sum of all those numbers is because the value of `x` is being replaced each time.)

See if you can come up with a solution for how to determine the sum of all the numbers you inputted. (Warning: the spoiler (answer) is in the margin.)

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

Two things to be aware of in doing this:

1. If you set the maximum number of items quite high and then get bored and need to exit the program – press the key combination Ctrl-C and **MATLAB** will exit your program (but leave **MATLAB** itself still running).

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

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

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

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

   ```
   x = 0;
   ```

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

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

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

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

   and then after the loop finishes, you can retrieve the sum of all the numbers by:

   ```
   sum(y);
   ```

   or if you like:

   ```
   sum(y(:));
   ```

   (Both give the sum of all the elements in `y`.)

 str='do you like bananas?';

2. If you run the program a second time and use the *vector* approach (see margin), something very odd starts to happen to the reported sum. This is because there already exists a *vector* with the same name left over from the first time you ran the *script* program. You can solve this (first try it out – running the program several times in a row to see what happens) either by initializing the vector `y`, just like you did for `x` in the 1st solution, i.e.

```
y = [];
```

(before the loop starts, of course), or you can clear the workspace using `» clear all` (clears *all* variables), or clear just the problem variable (`y`) that will end up growing and growing and growing ... (`» clear y`).
You could also add `clear all` to the very start of your *script* m-file.
Or ... you could make the *script* a *function*, making the vector created inside the *function* 'secret'.

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

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

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

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

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

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

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

Note that you should not re-use the same variable name `n` that you use for the *loop* counter, as in something like:

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

Why? (Try it and see, even.)

2.4.2 *Other loop configurations and usages*

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

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

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

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

and the all-important *function* header line would be:

```
function [] = function_sum(max_count)
```

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

```
» function_sum(5)
```

will request 5 inputs in turn, and at the end, display the sum.[38]

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

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

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

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

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

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

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

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

82 str='do you like bananas?';

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

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

2.4.3 *Fun(!) worked examples*

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

Loops, CAMERA, ACTION![39,40] (A more colorful example of *loops* in action.) What we are going to do is (load and) plot a sequence of monthly data-sets and put them together to create a movie (animated graphic) to illustrate the seasonality of temperature in global climate. You will hopefully thereby better appreciate the value of constructs such as *loops* in computer programming in saving you a whole bunch of effort and needless duplication of code. (Equally, you might not have wanted a movie as the end result, but simply a number of plots, all identical except in the specific array of data they were plotted from.)

[39] **Example codes provided**

[40] (at end of text)

First download all the monthly global surface temperature data-files on the course webpage (there are 12 files to download)[41]. Then you are going to want to plot them all ... which would get desperately tedious if you had to do this at the command line 12 times. Think how much more of your life you would be wasting if the data were weekly. Or monthly data for 1972 through 2003, some 372 separate data-files ... You would never have time to go get a coffee ever again(?) So we are going to use a *loop*.

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

To make an animation, we need to make a series of frames, with each one being a different monthly temperature plot (in sequence; Jan through Dec). The files are rather conveniently named: `temp1.tsv`, `temp2.tsv`, ... `temp12.tsv`[42]. We should start by loading this little lot in. For example, at the command line, to load in the the first file we could write:

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

```
» temp = load('temp1.tsv');
```

or equally:

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

Then for the 2nd file:

```
» temp = load('temp2.tsv');
```

and the third ...

```
» temp = load('temp3.tsv');
```

...

This is something that a *loop* could be used for while you go off for a coffee. So this is what we are going to do – use a *loop* to load in all of the files in turn.

Create a new *script* m-file. Call it ... anything you like[43]. 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. (If you get totally confused as to what should be going into the program and where – refer to the 'answers' for this section at the back of the book.)

We first need to construct the *loop* framework. We'll call the month number counter variable, `month`. Create a `for` *loop* (with nothing in it yet) with `month` (as the counting variable) going from `1` to `12`.[44] Refer to the course text (this document!), and/or the **MATLAB** documentation, and/or the entirety of the internet, if necessary. The syntax (and examples) is described in full under `» help for`. Save the script (m-file) and run it[45]. What happens? Can you tell?

One way of following what is going on as **MATLAB** executes the commands within a script is to explicitly request that it tells you how it is getting on. You can use the function `disp` to help you follow what the program is doing (this is Old School debugging[46]). Within the loop (anywhere!), add the following line:

```
disp(month)
```

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

We can construct filenames to load in by:

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

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

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

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

[43] `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.

[44] 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 %.

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

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

[47] 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.

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

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

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

and after it (still within the *loop*), add some some debugging[48]:

```
disp(filename)
```

just to confirm that appropriate filenames are being generated.

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

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

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

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

We'll now add some graphics.

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

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

(or could be simply written `pcolor(temp);`) and should produce extremely exciting graphics as in Figure 2.8[51].

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

`num2str`
Converts a number to a *string* (`s`), e.g.

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

where `N` is any number type variable.

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

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

[49] 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.)

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

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

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

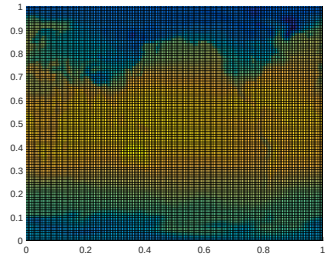

Figure 2.8: Extremely unappealing blocky plot of Earth surface temperature (who cares with month? – the graphics are too poor to matter ...).

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

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

you wont then end up with 24 figure windows. And then 36 the time after that, and ...

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

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

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

[53] Where to put the line? Within the loop and after you have plotted the plot.

This lines goes at the very end of the loop (before `end`), to give you:

```
for month=1:12
   filename = ['temp' num2str(month) '.tsv'];
   temp(:,:)  = load(filename);
   pcolor(temp(:,:));
   M(month) = getframe;
end
```

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

```
» movie(M,5,2)
```

and hopefully ... an animation of the progression of monthly surface air temperatures globally, should appear[54].

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

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

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

Creating a portable animation format would be useful (i.e. that you could play on a different computer or upload to the internet). There is no longer a convenient **MATLAB** command to turn the **MATLAB** format movie (`M`) into a format you can use elsewhere (there used to be a command called `movie2avi`, but it has been 'retired' (curse you, Mathworks)). The new/replacement command is `VideoWriter`, which differs mostly in that the animation is now

created within the program and the .avi format animation has its frames added (within the loop) as the graphics are created.

To use the **MATLAB** `VideoWriter` *function* to create the animation, you will need the following code (that you should put into a new m-file):

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

Note that if we wanted to save all the data in a 3D array, we could also have written:

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

and then for the 2nd month:

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

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

You could go on and load in the March, April, etc data in a similar fashion, but you should be able to see a pattern forming here – each filename differs only in the number at the end of its name and this number corresponds not only to the number of the month, but will also correspond to the layer index of the 3D array that you will create.

 str='do you like bananas?';

A loop for this might look like:

```
for month=1:12
   filename = ['temp' num2str(month) '.tsv'];
   temp(:,:,month) = load(filename);
   pcolor(temp(:,:,month));
   M(month) = getframe;
end
```

and at the end, you will have a 3D array – `temp` – containing all the months data.

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

No surprise that you might combine both *loops* and *conditionals* in the same programming structure. In fact, this becomes very powerful and is an extremely common device in programming. But this can all also become confusing ... remember to indent your code.

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

2.5.1 `for ...` *and conditionals*

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

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

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

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

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

Enter in the second code (or some variant of it) into a new *script* m-file, and explore how it works – try changing the alternative loop limits, add a line within the loop to `disp` the value of `n` and hence confirm that the correct number of iterations of the loop occurs. Note that you will need to replace `%` with your own line (e.g. using `disp`) or you could have nothing and leave the comment % line in. Remember to add comment lines.

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

```
% set up some strings for the 2 questions
my_question1 = 'Please enter a number:  ';
my_question2 = 'Another input (y/n)?  ';
% initialize running sum
running_sum = 0.0;
% START OF LOOP
for n = 1:1000000
   % ask for a number
   my_number = input(my_question1);
   % update running sum
   running_sum = running_sum + my_number;
   % display running sum
   disp(['sum so far = ' num2str(running_sum)])
   % ask whether to keep going
   my_string = input(my_question2,'s');
   % exit loop if answer is no
   if strcmp(my_string,'n')
       break
   end
end
% END OF LOOP
```

where `1000000` in the code is simply chosen as a 'very large number' and one rather larger than the maximum number of numbers you could ever imagine entering[55].

The key new command here is `break`. The way the code works (hopefully!) is that towards the end of the *loop*, the 'another input' question is asked – if no further input is required, the loop exits via the `break` command. Remember that now we have *loops* and *conditionals* nested together, it helps even more to *indent* the code as per illustrated in the given code fragment[56]. Also note that here – the two different questions (demands) outputted to the screen – 'Another input (y/n)?' and 'Please enter a number' – are pre-defined before the *loop* starts. These same text could equally be placed directly within the *loop* within the call to the `input` *function*.

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

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

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

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

```
...
for n = 1:1000000
   ...
   if ~strcmp(my_string, 'y')
       break
   end
end
```

which compares `my_answer` and 'y', if this is not true (that they are the same), `break` is executed. (Note that many of the lines of code from before have been omitted (`...`) for brevity.)

A PRACTICAL EXAMPLE of testing the value of a *variable* and `break`-ing out of a *loop* depending on the result of the test, would be when saving a data file. You might test for a filename that already exists and if so, automatically modify the new file name so as not to overwrite the existing file.[57] The relevant function is `exist`, and in the case of a test for a file, the function returns either `0` (the file does not exist in the **MATLAB** search path, although that does not rule out it existing somewhere else entirely), or `2` (the file exists).

Clearly(?), in the example of saving the movie file, you might well want to test whether the filename that you have chosen already exists (i.e. the value returned by `exist` is `2`). If so (i.e. the file exists), you need to modify the filename by means of a new concatenation, perhaps appending something like '`_NEW`' to the end of the string[58]. If not, and the filename has not already been used, you can proceed as before – the equivalent of 'doing nothing'.

Make a copy of the avi movie code you were given in the previous Section (you can also find this in Chapter 7.2 under '*Code for creating an avi format animation*'). You are going to modify this so that at the very start, it checks to see if the particular filename has already been used.

In modifying the code, you could start by defining a default filename[59] that you will use if there is no clash with any existing file, e.g.

```
my_filename = 'my_animation.avi';
```

Now, on the next line, test whether this filename already exists:

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

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

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

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

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

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

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

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

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

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

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

[59] At the very start of the program and just before the `% Prepare the new file` comment line.

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

where here, we take the string contained in `my_filename`, we append a 'NEW_' to the start[60], and assign the new (longer) string back into the variable `my_filename`. The complete code addition will then look like:

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

```
my_filename = 'my_animation.avi';
if (filename_check == 2)
   my_filename = ['NEW_' my_filename];
else
   % DO NOTHING
end
```

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

The only change to the existing code you need to make, is this line:

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

because you no longer want to use the same default filename each and every time you run the animation, but rather, pass the variable containing the filename:[61], i.e.

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

```
vidObj = VideoWriter(my_filename);
```

(remembering that you do not put the variable name in inverted commas).

Make sure that the order of the required lines of new of code is:

1. Set default filename.
2. Test whether this filename already exists and assign to the variable `filename_check`.
3. Test whether the value of `filename_check` is `2` and if so, modify the filename.
4. (Edited `vidObj =` line.)

2.5.2 `while ...`

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

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

Here – `STATEMENT (IS TRUE)` is the conditional. For instance and rather trivially, **create the following as a new *script* m-file and run it**[62]:

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

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

What has happened is that true is always ... true. Hence the condition is always met and the `while` *loop*, loops forever. Conversely, `while false` would never *loop*, not even once – try it:

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

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

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

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

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

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

then the *loop* will execute exactly 10 times (just as per `for n = 1:10`) (**try this**).

You could also do the counting in reverse:

```
n = 10;
```

 str='do you like bananas?';

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

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

It is not always completely obvious whether even simple while loops like this execute 9 or 10 (or 11) times particularly when often you might come across `while (n >= 0)` that allows the loop to continue when when `n` has reached z value of zero (but not below). **Spend a little while playing about with different `while` configurations and *loop* criteria, adding `disp` lines or *breakpoints* to find out how many times the *loop* executes in total.**

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

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

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

FINALLY ... WE COULD UPDATE THE FILENAME CHECKING EXAMPLE ... USING WHILE. The problem with the previous code is that you checked for the existence only a default filename (and appended '_NEW' if a file already existed).

One (partial) solution would have been to, rather than append a pre-defined string ('_NEW') to the filename, request that the user provide a completely new filename.

A complete solution would be to address the situation when asking for an alternative filename ... if that file existed too. We could

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

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

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

```
filename_check = 2
```

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

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

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

 str='do you like bananas?';

2.6 *Even more (and loopier) loops*

[Further examples of increasingly extreme loopiness.]

[**OPTIONAL**] Looping through arrays. In plotting e.g. global temperature distributions, it would be nice to add on the continental outline on top. Currently, and particularly with the very basic 2D plotting you have seen so far (`pcolor`), you are to some extent left guessing where the land and where the ocean is. We are going to work through using a loop to process some data that defines a series of line segments that make up the outlines of the continents.[63]

[63] **Example codes provided**

The first 2 files (that can be downloaded from the website), comprise a series of pairs of lon-lat values that delineate the outline of the continents and all but the smallest of islands:

- continental_outline_lat.dat (labelled 'lat')
- continental_outline_lon.dat (labelled 'lon')

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

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

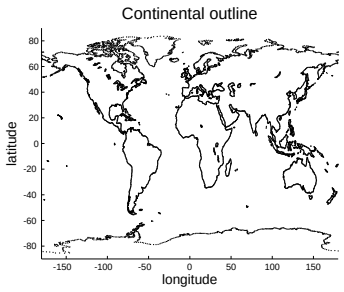

Figure 2.9: Continental outline (of sorts).

Your *script* **m-file**) for this might look like:

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

(but with lots of comment lines!).

By plotting dots (points), the coastal outline at higher latitudes gets increasingly pixelated. So, we might instead plot as lines between the lon-lat pairs. For this, you could simply use `plot`.

Copy your m-file (make sure it was saved first!), rename it, and then edit it to use `plot` instread of `scatter`. You should end up with something like Figure 2.10..

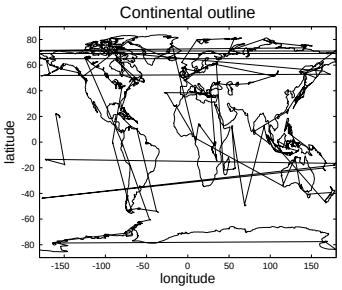

Figure 2.10: Another continental outline (of sorts).

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

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

- continental_outline_start.dat (labelled 'start')
- continental_outline_end.dat (labelled 'end')

i.e.:

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

(Note that you cannot simply call the second variable `end`, because **MATLAB** is already using it as a special word.)

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

Well ... this is probably about as unexciting as it gets – a small piece of the Antarctic coastline. If you do a `hold on` and plot the next block (rows `101-200`), you'll get the next chunk of coastline:

```
» plot(lon(101:200),lat(101:200));
```

You could keep going this – manually adding additional sections of the global continental outline. This could get tedious ... and it turns out that there are 283 different fragments to plot, all one after another. (This number comes from asking **MATLAB** the `length` of `lstart` or `lend`, e.g. `length(lstart)`) This is, of course, why we

`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).

need to get clever with a *loop* and automatically go through all 283 fragments, plotting them on on top of another in the same figure.

How? First you need to write the `plot` command in a more general form – you do not want to have to read the values out of the `lstart` and `lend` vectors 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)[64]. Re-writing again, for the first fragment, this looks like:

```
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.

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 could 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[65])?

So ... **create yourself a new m-file** (and copy-paste whatever you like from the previous *script* tha saves you time)in terms of loading in the data. At the start – load in the lon-lat pairs as vectors (renaming then to something more manageable if you wish) and then load in the vectors containing the start and end information.

After that in the code – create a `do ... end` loop. Maybe before you try and plot anything, print (`disp`) the loop count and run the program (after saving), just to check first that the loop is functioning correctly.

Also before the loop starts, create a `Figure window.` and set `hold on`. You now have a basic shall of a program – loading in the data, initializing a figure, and appropriate looping, but not yet actually doing anything within the loop.

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

Run it. Hopefully ... something like Figure 2.11 appears(?)

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

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

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

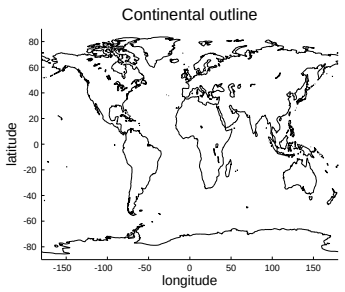

Figure 2.11: Another go at the continental outline!

3
Further ... MATLAB and data visualization

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

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

3.1 *Further data input*

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

Please regard this section as a simple overview on some of the different **MATLAB** file loading/input options. At this stage, it is more important that you are aware of different data formats and the various ways of importing them into **MATLAB**, then *per se* having to master all the details of using them. For many one-off data loading problems, it can be easier to use the **MATLAB** GUI and the Import Data wizard.

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

opening and closing files

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

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

3.1.1 *Formatted text (ASCII) input*

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

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

`textscan`

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

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

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

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

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

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

(see Box). You are going to have to pass the ID of the file again when you call this *function* (so **MATLAB** knows which file to close).

4. Lastly, you are going to have to deal with the special data structure that **MATLAB** has created for you ...

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

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

OK – so having seen the format of the data in the ASCII file, you are going to work through the following steps (you can do all this at the command line, or add the lines one-by-one to a *script* m-file if you prefer)[3]:

1. First 'open' the file – you will be using the *function* `fopen`, and passing it the filename[4] (including the path to the file if necessary, i.e. if the **MATLAB** working directory does not also contain the file). So that you can easily refer to the file that you have opened later, assign the output of `fopen`[5] to a variable, e.g.

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

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

   (a) The ID of the open file (you have assigned this to a handy variable (`openfile_id`) already.)
   (b) The *format* of the file (see margin note).(This is where it gets much less fun, but hang in there!) You simply list, space-separated, and between a single set of quotation marks, one format specifier per element of data.
   In this particular Example, there are 4 items of data (per row) – each of them is either an integer or a floating point number[6], depending on how you want to look at it. Assuming that the

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

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

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

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

data is a floating point number, the *format* for the input of each number item, is `%f`.

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

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

Here, the `'%f %f %f %f'` bit specifies that the data format consists of 4 floating point (real) numbers.

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

```
» fclose(openfile_id);
```

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

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

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

To help you – a complete code for doing all this is given at the end of the book.

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

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

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

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

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

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

As a further example, we are going to process a more complicated version of the paleo atmospheric CO_2 proxy dataset. This file is called paleo_CO2_data.dat (rather than .txt) and is again available from the course webpage. An initial problem here is even opening up the file to view it – if you use standard **Windows** editors such as **Notepad** it fails to format it properly when displaying its contents[7].

The first lesson then in scientific computing then is to have access to a more powerful/flexible editor than default/built-in programs such as **Notepad**. One good (**Windows**) alternative is **Notepad++**.[8]

If you can open the file with something like **Notepad++** (or are in any case using a **Mac**) – first note the format – there are a bunch of header lines and moreover, some of the columns are not numbers (but rather strings). Even if you were to manually edit out the headers by adding `comments` (%)[9], you are still left with the problem of mis-matched columns. You could edit the file in **Excel** to remove the problematic columns as well ... but now this seems like a real waste

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

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

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

of time to be editing data formats with one software package just to get it into a second! (Again, you could use the **MATLAB** GUI import functionality ... but it will be a healthy life experience for you to do it at the command line :o))

OK – so having gotten an idea of the format of the ASCII data file, you are going to tackle this in the same 4 steps as before:

1. First 'open' the file as you did previously (using `fopen`) and assigned the ID returned by the `fopen` *function* to the variable: `file_id2`.
2. Call `textscan`. However, we now want to pass 3 pieces of information (compared to 2 before):
   (a) The ID of the open file.
   (b) The *format* of the data.
   (c) And now – a parameter, together with an (integer) value, to specify how many rows of the file are to be assumed to be header information and hence skipped.

(Again – the result of `textscan` is then assigned to a variable which will represent a *cell array*.)

Lets do the easy bit first – to tell **MATLAB** to skip *n* lines of a file. For this – add the parameter `'HeaderLines'` to the list of parameters passed to `textscan`, and then simply tell it how many lines to skip. In this example:

```
my_data = textscan( ...
     file_id2, ...  ,'HeaderLines',3);
```

will tell **MATLAB** to skip the first 3 lines of the file.

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

```
my_data = textscan( ...
     file_id2,'%d %d','HeaderLines',3);
```

Try it (including closing the file, and a call to `cell2mat`, as before). What has happened?

It seems that **MATLAB** translates your format (`'%d,%d'`) into: 'read in a pair of integers, and keep automatically repeating this, until something else is encountered'. That something else, is sequence of characters at the end of the first data line (line #4, because we skipped the first 3), that makes **MATLAB** think that it has finished (or rather, that it cannot reading in 2 pairs of integers any longer). This leaves you with 2 pairs of integers – i.e. a 2×2 matrix (as you'll see if you look at `my_data_array`).

Here is a solution – we could omit all the information following the first 2 elements (something for **Google** to help with).[10]:

[10] 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]`).

 str='do you like bananas?';

```
my_data = textscan( ...
     file_id2,'%d %d %*[^\n]','Headerlines',3)
```

The weird bit here translates to ... `%*` == ignore field ... until the line end == `[^\n]`, and then read repeat for the next line.

(You are not expected to know or remember this nor be tested on it ... just park all this at the back of your mind and that there are flexible ways of dealing with data input, including not necessarily reading everything in!)

3. Now you can close the file:
   ```
   fclose(file_id2);
   ```
4. ... and convert the results to something human-readable:
   ```
   my_data_array = cell2mat(my_data);
   ```

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

The complete code (minus all the lovely comments!) looks like:

```
file_id2 = fopen('paleo_CO2_data.dat');
my_data = textscan( ...
     file_id2,'%d %d %*[^\n]','Headerlines',3);
fclose(file_id2);
my_data_array = cell2mat(my_data);
```

(remembering that the `...` just indicates a break in what is otherwise a continuous single line).

Put this into a new *m-file*, comment it, so that you know what all the bits are doing (and what the overall program itself does), and try it out. NOTE: if you copy-paste from the textbook PDF ... be careful not only to replace the PDF inverted comma character with a 'normal' (**MATLAB**-friendly one), but the ^ symbol can also come out incorrectly [11] ...

(There must be some sort of important life lesson hidden here in all this. Perhaps about only working with well-behaved data files, or using the GUI import functionality?)

[11] The symptom being that only the first line of data is read in.

3.1.2 *Importing ... Excel spreadsheets*

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

1. Select some, or all, of the columns and rows in a specific worksheet, and either copy-paste this into a text file (but taking care that the worksheet column widths are formatted such that they are wider than the widest data element), or save in an ASCII format, with comma or tab delineations between columns. In either

case, then load in the data using `load`, or if consisting of mixed numbers/text, go through the Hell that is `textscan` ....

2. Use **MATLAB** *function* `xlsread`.

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

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

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

Note that the simple usage of the `xlsread` *function* gives you an *array* containing just the numeric data. If you were to type:

```
» [num,txt,raw]=xlsread('paleo_CO2_data.xlsx');
```

then you still get the numeric data returned in the array `num`, but you also get 2 *cell arrays*[12] – `txt` and `raw`. The *cell array* `txt` contains just the text data, and `raw`, 'everything'. View these in the Variable window (by double-clicking on the variable names in the **MATLAB** Workspace window). NOTE that **MATLAB** takes some time to open and process **Excel** format files and the command will not complete as quickly as e.g. `load`.

If you happen to have an **Excel** file with data (of any sort) in it (e.g. from another class), practice loading in its contents into **MATLAB**. Note that if the **Excel** file contains cells with text in and you want the text data, then you'll need to use the more advanced format of `xlsread` (see Box or `help`). Also try loading into only a single sheet of an **Excel** file (assuming that the file has multiple sheets).

`xlsread`

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

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

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

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

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

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

```
» A(4,3)
```

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

```
» C(4,3)
```

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

```
» C{4,3}
```

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

3.1.3 *Importing ... netCDF format data*

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

MATLAB actually has a quick and simple (concentrate on this first one!), and ... a complicated long-winded formal way (simply note the existence of this one!) of accessing data in a *netCDF* file:

1. **Using `ncread`, which reads data directly from the file.**

   `ncread` is by far the simplest way, although it lacks in flexibility and deviates from standard practices used across other programming languages.

   As an example, we'll take the output of a low resolution climate model. To start off, download the 'model netCDF file – 2D' *netCDF* file – fields_sedgem_2d.nc. The data here is relatively simple – a 2D distribution of bottom-water and surface sediment properties, saved at a single point in time. In other words, there are only 2 (spatial) dimensions to the data[13].

   OK – we'll start by opening the file. The ID of the variable we want to extract and plot is called 'grid_topo'. To load/extract the 2D field and assign it to a variable `data`:

   ```
   » data2d = ncread('fields_biogem_2d.nc','grid_topo');
   ```

   You should now have an array called `data`. It should be 36×36 in size. **Plot it**[14]. Can you deduce what it might be of? Is it in the correct orientation? (If not – correct it, by rotating the array, and/or flipping the rows or column.)

   (There are more appropriate ways of plotting this, which we will encounter later.)[15]

   The variable names of other data-sets that you might load (in place of passing `'grid_topo'`) and experiment with in terms of plotting function, color scale, and any other refinements that help visualise the data, include:

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

2. **Via a series of function calls to the netCDF library.**
   In the formal and more long-winded approach, you open the file and receive an ID for that file. The file can then be written to or read (including just interrogating its properties rather than necessarily extracting spatial data) using this ID. And of course, closed (using the same ID). The *netCDF* standard is also little odd in how reading/writing is implemented and everything has to be done by determining the ID of a particular data variable or property of the file. The general approach is as follows:

`ncread`
In its simplest incarnation:

```
data = ...
ncread(filename,varname)
```

where `filename` is the name of a *netCDF* file, and `varname` is the name of the data variable in the *netCDF* file.

e.g. if there was a variable called `rain` in the file `climate.nc`,

```
data = ...
ncread('climate.nc','rain')
```

would read the values in the *netCDF* file variable `rain` and assign to the variable `data`.

MATLAB provides a couple of further tricks, allowing you to read sections of the full *netCDF* variable data array, or sample the data array – see **help**.

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

[14] Your choice of 2D plotting function that you have already come across, e.g. `pcolor` or `image`, although not all work as well on this particular dataset (e.g. the auto scaling in `image` causes issues).

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

(a) Open the netCDF file by

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

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

(b) This is the weird bit, as we cannot ask for the data we want automatically :o) Instead, given that we know[16] the name of the variable we want to access, we ask for its ID ...

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

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

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

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

So actually, not totally weird – you request the ID of the variable, then use that to get access to the data itself. The names of the **MATLAB** commands vaguely make sense in this respect – `inqVarID` for inquiring about the ID of a variable, and `getVar` for getting the variable (data) itself[17].

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

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

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

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

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

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

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

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

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

in which we obtain the ID of the axis variable `'lat'`, then retrieve the axis data and assign it to a vector `lat` (and then likewise for longitude). One could then take the 2 axis vectors, and create a pair of matrices – one containing longitude values associated with the 2D data points, and one containing latitude values associated with the 2D data points. For this, you would need to use the function `meshgrid`. (We'll re-visit this example once you have seen `meshgrid` in action.)

[**OPTIONAL**] We can extend the visualization problem to 3D – 2 spatial dimensions (longitude and latitude) and one of time. The file you will need to download to experience these wonders, is called fields_biogem_2d.nc

To load the variable 'atm_temp':

```
» data3d = ncread('fields_biogem_2d.nc','atm_temp');
```

How many dimensions does this array have (e.g. use `size`, or ensure that the Size column in the Workspace window is selected)? What are the lengths along each dimension? Can you deduce which of the dimensions, time might be?

Plot a lon-lat slice. Note that you need to select all longitudes and all latitudes in the array, but only one time index.

Finally – to test your understanding to date, create an animation of how the surface air temperature in the model evolves over time.[18]

[18] You have everything you need – the *vector* of years, and from this you can determine how many different time points (and 2D data slices) there are, and hence the number of iterations of a *loop*.

3.1.4 *Importing ... with* ***Import Data***

Finally, there is also a GUI icon Import Data and a little like Excel and gives you the option to specify the number of header lines to ignore etc. Try playing with this for any of the files imported in this section (except the netCDF one).

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

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

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).

3.2.1 *Contour plotting*

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

For a feel of what you should be able to learn to achieve using **MATLAB** – go to the following webpage. In this data repository you can do things like re-plot with different longitude, latitude, and temperature ranges. Overlay the coastlines, and other useful things like that. You can also click through the different months of the year to get a feel for how the surface temperatures on Earth change with the seasons. (However, the graphic produced from this particular website is not particularly great, and you will learn to do at least as good as this!)

As an example, load in the '*global Earth surface topography*' 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[19].

After loading the data file into the **MATLAB** workspace, try

[19] 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.

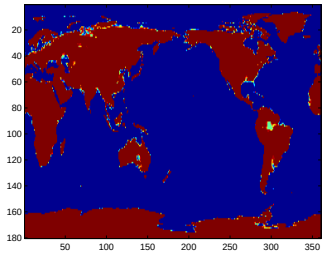

Figure 3.1: Very basic imaging (`image`) of an array (2D) of data – here, global bathymetry.

throwing the array into the `image` function (which you saw previously) see what happens, e.g.

```
» image(etopo1deg);
```

(hopefully something like Figure 3.1, but perhaps with a different default color scale).

If it had happened to come out displayed upsidedown[20], then you'd need to first (before plotting using imagesc) flip the matrix upsidedown using the command:

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

(and then re-plot using `image`), and if the Earth instead appeared on its side you will need to swap the rows and columns (x for y axis):

```
etopo1deg=etopo1deg';
```

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

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

```
» imagesc(etopo1deg);
```

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

HINT: You can use the function `max` to determine the maximum value in a vector, and for the entire array:

```
etopo1deg_max=max(max(etopo1deg));
```

as we did for sum in the first chapter, or more elegantly:

```
etopo1deg_max=max(etopo1deg,[],'all');
```

(refer to **MATLAB** `help` for the syntax and why the `[]` is needed), and hence finallly to:

```
» imagesc(etopo1deg,[0.0 etopo1deg_max]);
```

Which sort of in a round-about sort of way also brings us to how to set the color scale, which can be changed using the `colormap` command (see Box).

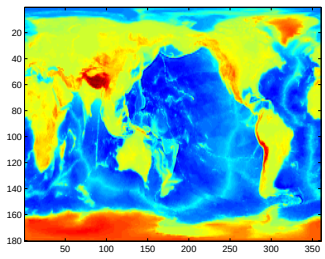

Figure 3.2: Slightly improved very basic imaging (`imagesc`) of bathymetry data.

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

imagesc

For a data array (matrix) `A`,

```
imagesc(A)
```

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

To set the limits of the color scale, e.g. from (z) value of 0.0 to 10.0, you can pass the limits as a vector:

```
imagesc(A,[0.0 10.0])
```

(and very similar to setting the x- or y-axis scales).

colormap

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

```
» colormap NAME
```

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

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

To return to the default *colormap*:

```
» colormap default
```

At the command line, try out setting a different *colormap*, e.g.

```
» colormap 'pink'
```

and re-then re-plot the global topography data. Try out various different color maps/scales (» `help GRAPH3D` will give a list of **MATLAB** colormaps). What color and (min,max) scales work well and what do not? Which scales help pick out details of e.g. ocean floor depth variation and which help pick out simple land-sea contrasts. Think about what one might want to highlight about global topography and what color scale might be best for this purpose?

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

To start with, you can simply use the `contour` function (see Box), passing only the matrix (of global topography values). Try this, e.g.

```
» contour(etopo1deg);
```

Now you might want to think about flipping the matrix up-down, and/or left-right, as your plot should have come out looking like Figure 3.3 (depending on your chosen colormap) and may need adjusting.

Once you have fixed the orientation of the topography map, you might play about with the color scale (`colormap`) as before. You might also try the companion to `contour`, called `contourf` which gives you something like Figure 3.4, e.g.

```
» contourf(etopo1deg);
```

OK, so a next refinement in plotting esp. maps and contour plots, is firstly, to specify the range of the color scale, as we may not want the min-to-max range chosen by default by **MATLAB**, and then, to control the number of contours (e.g. in the topography example, they are pretty far apart and it is difficult to make out much detail). Both of these factors can be addressed simultaneously, by giving **MATLAB** a vector containing the value at which you want the contours drawn[21].

Taking the global topography data – lets say you were interested only in low lying and shallow bathymetry, and wanted 20 contours intervals. Assuming a range in topographic height (relative

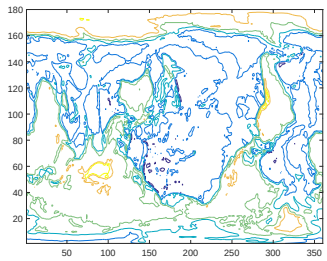

Figure 3.3: Example result of basic usage of the `contour` function.

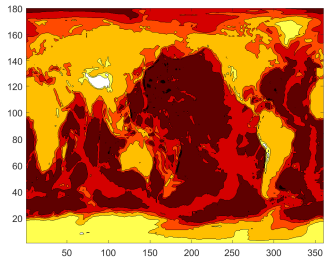

Figure 3.4: Example usage of `contourf`, with the `hot` *colormap* (giving dark/brown colors as deep ocean, and light/white as high altitude).

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

to sealevel) of -1000 m to +1000 m, you should be able to deduce how to create the vector(?)[22] Do this and check e.g. by opening up the vector in the Variables window. You should see the numbers from -1000 to 1000 in intervals of 100.[23]

Having created a specific vector of contours to plot, try it out, by:

```
» contour(etopo1deg,v);
```

(see Box for syntax).

OK – so this is a little weird and maybe not so useful, but you get the point, hopefully ... Try (still at the command line) plotting some or all of the following:

1. Contours of topography from sealevel, to 10,000 m, in increments of 100 m.
2. Just (on its own!) the sealevel (coastline) contour ... trickier – create a vector with a value at zero, and a value either side – one very high and one very low. Use `contour` rather than `contourf`, although the latter produces a lovely land-sea mask!
3. Convert the data matrix of value in units of m, to ft, and plot the ocean floor (values equal to or below sealevel) in intervals of 1000 feet (e.g. `v=[-20000:1000:0]`). (HINT: you'll need to re-scale the data array using the conversion: $1m = 3.28084ft$)
4. Finally – try some different color scales for the above. Think about which color scales best help illustrate the data, and whether `contour` or `contourf` is clearer. Also: how many contour intervals is 'best'? You key is to make features clear, within the plot becoming cluttered or overly detailed.

The final refinement in contour plotting we'll look at here is adding labels to the contours. The command to do this is `clabel` (for 'contour label') (see Box). Now, before anything, there is a slightly complication. `clabel` needs to know details of the contours and graphics object with which to do anything with. For the purposes of this course, you don't have to worry about the details of this ... but simply need to note and remember the following:

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

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

   which saves a matrix of data to variable `C`, and a ID (technically: graphics object 'handle') to variable `h`.
   You can test what has been returned by typing:

[22] If not, it is:

```
» v = [-1000:100:1000];
```

[23] Why, for instance, can you not simply write:

```
» v = [-1000:1000];
```

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

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

```
contour(Z)
```

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

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

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

More involved and practical, is:

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

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

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

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

`clabel`

```
» clabel(C,h)
```

labels every contour plotted from

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

(or from `contourf`).

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

```
» clabel(C,h,v)
```

```
» [C,h] = contour(etopo1deg,v);
```

and looking to see what new variables, if any, have appeared in **MATLAB** Workspace.

2. When you call `clabel`, pass these parameters back in, e.g.

   `clabel(C,h)`

(in its most basic usage). So in the example above:

```
» clabel(C,h);
```

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

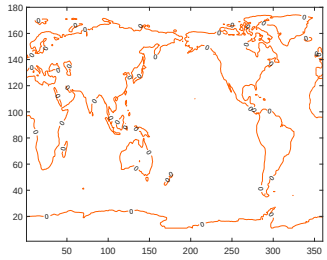

Figure 3.5: Example usage of `contour`, contouring only the zero height isoline, and providing a label.

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

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

(which creates 100 m spaced contours between -1000 and +1000 m) maybe you only want labels on contours every 500m, so you'd create a different vector:

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

to specify the labelling intervals. The complete set of commands for this becomes:

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

where `[C,h] = contour(etopo1deg,v);` specifies to contour from -1000 to 1000 in steps of 100, `but clabel(C,h,w);` says to label only every 500 m (from -1000 to 1000).

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

```
» colorbar
```

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

`colorbar`

This almost could not be simpler:

```
» colorbar
```

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

```
» colorbar(PLACEMENT)
```

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

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

```
» h = colorbar
```

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

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

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

3.2.2 *Meshgrid*

We'll now address the issue with missing/incorrect lon/lat axis labels on the plots.

Each data point in the `etopo1deg` matrix should have one longitude value (x-axis) and one latitude (y-axis) value associated with it. What we need, is a pair of matrices, of exactly the same size as the `etopo1deg` data matrix – one holding longitude values and one latitude values.

There are various ways of creating the required matrices 'by hand' (or involving writing a program including a *loop*). All of them are tedious. There is a **MATLAB** function to help. But it is not entirely intuitive[24] ... `meshgrid`.

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

```
» [X,Y] = meshgrid(1:3,10:14)

X =
     1  2  3
     1  2  3
     1  2  3
     1  2  3
     1  2  3
Y =
     10  10  10
     11  11  11
     12  12  12
     13  13  13
     14  14  14
```

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

In our example 2D (topography) dataset – start by noting that the topography data is on a regular 1 degree grid starting at 0° longitude. Latitude starts (at the bottom) at -90° and goes up to +90° . We need a matrix containing all the longitude values from 0° to 359° and latitude from -90° to 89° .[25] These matrices need to be the same size

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

`meshgrid`
The unholy syntax is:

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

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

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

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

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

as the data matrix.

Maybe just go ahead and do it right now ... and then pause and understand what has happened after. Create the longitude and latitude grids by:

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

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

The only way to fully make sense of things now, is to use it. Remember that use of `contour` (and `contourf`) can take matrices of x and y (here: longitude and latitude) values that correspond to the data entries in the data matrix (`etopo1deg`).

Re-load the topography data in case you have flipped it about in all sorts of odd ways, and then do:

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

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

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

or if you prefer 2 explicit steps:

```
» etopo1deg_corrected = flipud(etopo1deg);
» contour(lon,lat,etopo1deg_corrected);
```

Phew! (Figure Figure 3.6.)

The final complication (but don't worry about actually doing this) is that the data points in the gridded dataset (matrix `etopo1deg`), technically correspond to the mid-points of a 1 degree grid, not the corners. So if we were going to try and be formally correct[26], our vectors that we'd pass into `meshgrid`, would be:

```
» x = [0.5:359.5];
» y = [-89.5:89.5];
```

and hence:

```
» [lon lat] = meshgrid(x,y);
» contour(lon,lat,flipud(etopo1deg));
```

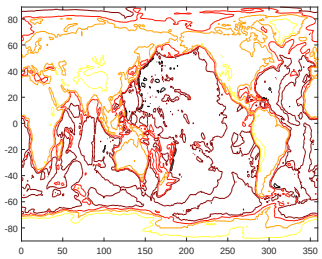

Figure 3.6: Usage of `contour` but with lon/lat values created by `meshgrid` function and passed in (and with the `hot` *colormap* (giving dark/brown colors as deep ocean, and light/white as high altitude).

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

 str='do you like bananas?';

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

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

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

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

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

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

Place this code somewhere before the loop starts in your program.

Now use the longitude and latitude values matrices, in conjunction with `contour(f)`, to plot the global temperature distributions 'properly', e.g.

```
contour(lonv,latv,temp);
```

or if you prefer:

```
contour(lonv(:,:),latv(:,:),temp(:,:));
```

and it helps you remember which variables are arrays.

Try plotting just one plot first (e.g. by adding a breakpoint at the end of the loop, i.e. the line with `end`), before looping through all 12 months.

At this point (before creating an animation), you might also explore some of the plotting refinements we saw earlier. For example,

as per Figure 3.7. Firstly – get the units of the temperature data array into units of °C (or °F if you are into that sort of thing) rather than °K. Either: assign the `temp` array data to a new array and make the appropriate conversion from °K (all within the loop), or you can do this subtraction on the line that you actually plot the data (i.e., within the `contour/contourf` function), for example:

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

would convert to °C as it plotted the data.

You can also (assuming you converted to units of °C) set the plotting temperature limits and contouring consistent between months and with greater color interval resolution by adding the following line (before the loop starts):

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

and then to the `contour(...)` (or `contourf(...)`) function, add to the end of the list of passed parameters – `v`, e.g.

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

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

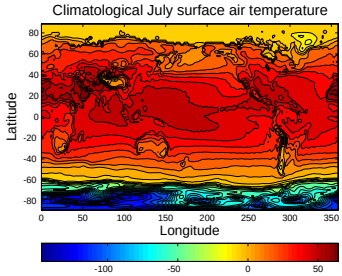

Figure 3.7: Example contour plot including `meshgrid`-generated lon/lat values. Result of `contourf(lon,lat,temp7,30)`, where the data file was `temp7.tsv`, with some embellishments.

Lastly – for any (or all) of the Examples above, you could add the continental outline to the plot. Remember, to use `hold on` in order to overlay the continental outline on top of the contour map without replacing it in the Figure window.

It should be obvious that plotting the continental outline might be something you want to us more than once. Sections of code that might get used multiple times are commonly placed in a file (or special section of the file) of their own and *called* from the main program that needs it. For example, you could place the entire continental outline plotting code, including loading in the data, in an m-file and make it a *function* – in this case, taking no parameters as input, and return no output.[27] In the example of the looped animation, in the sequence of code (within the loop), you will need to *call* your continental outline plotting function just after you have plotted the contour (or bitmap) plotting function.

[27] Make sure that you do not open a figure window with the `figure` command with the function, or you will not get a continental output overlay on your plot, but rather a sperate Figure window with just the continental outline on.

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

```
118  str='do you like bananas?';
```

3.3 *Further data processing*

This section contains a selection of further simple techniques for doing useful stuff with data, as well as for better graphing.

3.3.1 `find!`

So – a single **MATLAB** function gets its own sub-section, all to itself. Either it's really powerful and useful, or I am running out of ideas for the textbook[28].

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

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

OK – some Examples.

Say that you have a *vector* of numbers, e.g.:

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

and you want to find the maximum value in the vector – easy[29].

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

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

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

We can start by determining the maximum value (in the array, `A`).

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

Now, we use `find` to evaluate where (what index) in array `A` the element with a value of `max(A)` (equal to `c`) occurs. The following should accomplish this:

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

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

Finally, if we assign the result of `find` to `d`, remembering that `find` return an array index (or indices), we can then use `d` to determine the time at which the value of `9` occurred, i.e. `B(d)` which evaluates to `4` (whatever units of time):

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

[28] Answer: it is really powerful and useful.

`find`

MATLAB defines find, with a basic syntax of:

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

as *'return[ing] a vector containing the linear indices of each nonzero element in array X'*.

That means ... nothing to me. This is going to have to be a job for some Examples ... (in order to see what `find` is all about).

Actually, `find` returns the indices of the non-zero elements in the array and if the array is a vector, what is does is simple. For a matrix, **MATLAB** counts the elements sequentially, starting at the 1st row and 1st column, and working down the first column, rather than provide the (row,column) for indexing format you are used to. Hence where the *'linear indices'* bit comes in.

Furthermore, 'non-zero' indices is really just code-word for 'true'. So you are asking where the true values occur in `X`. If `X` is the answer to a *logical* or *relational* operation, then find tells you the indices of the elements that are true.

For example, `3 > [5 3 1]` equates to `[0 0 1]`, i.e. only the first element in the vector `[5 3 1]` is less than `3`. Hence:

```
find(3 > [5 3 1])
```

first evaluates the relational operation and generates a vector of true and false values, and then `find` tells you the index (or indices) where the true values occur (here, `ans = 3`).

`min`
`max`

Return the minimum and maximum, respectively, values in an array. e.g.

```
min([4 8 3 1])
```

will return a value of `1`.

[29] I hope so ... check back earlier (or slightly later) in the course on `max`.

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

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

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

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

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

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

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

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

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

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

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

```
» find(A(:)>5)
```

```
ans =
     2
     5
     6
```

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

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

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

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

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

Note that the size[30] of A has now shrunk to 5 – all the other elements remain, and in order, but the elements with a value greater than 5 have gone. You could apply an identical deletion (filtering) to the time array (`B(d) = []`).

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

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

For an Example of data-filtering – dig out the paleo-proxy (`paleo_CO2_data.txt`) atmospheric CO_2 data you downloaded earlier. One further way of plotting with `scatter` is to scale the point size by a data value. We could do with by:

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

... except ... it turns out that there are atmospheric CO_2 estimates of zero or less and `scatter` will refuse to scale the point size by such values ...

This leads us to a new use for `find` and some basic data filtering. We'll start by tackling the zeros.

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

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

`NaN`

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

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

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

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

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

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

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

What this is saying is: first find all the locations (row indices) in the 2nd column of `data` for which the value is equivalent (`==`) to zero. Then, replace the CO_2 value in all these rows of the 2nd column (which is originally zero) with a `NaN` (technically speaking: assign a value of `NaN` to these locations).

You have now filtered out zeros, and replaced the offending values with a `NaN` and when **MATLAB** encounters *NaNs* in plotting – it ignores them and omits that row of data from the plot.

Alternatively, we could have simply deleted the entire row containing each offending zero.[31] 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[32], you should notice that the size of the array data has been reduced (by 4 rows, which was the number of times a zero appeared in the 2nd column).

We are almost there with this example except it turns out that there is a CO_2 proxy data value less than zero(!!!) We can filter this out, just as for zeros. I'll leave this as an exercise for you – remember the operator 'less than' (`<`).[33]

The plot should end up looking like Figure 3.8. As another lessonette, given that the circles are insanely large ... you might try plotting this with proportionally smaller circles, which you could achieve by e.g.

```
» SCATTER(data(:,1),data(:,2),0.5*data(:,2))
```

which simply make the 2nd column values passed to `scatter` to set the marker size, to half their original value.

Having inserted `NaN`s into an array, or having ended up with `NaN`s in an array for other reasons, you can also search for (`find`) the `NaN`s. The first thing to note in looking for `NaN`s, is you cannot test for a `NaN` with a simple equality operator:

[31] First: Re-load the paleo-proxy atmospheric CO_2 data so that you can have another go at filtering it.

[32] Or:
```
» size(data)
```

[33] Actually, you could have done the filtering of both zeros, and values less than zero, all in one, using: $\leq$

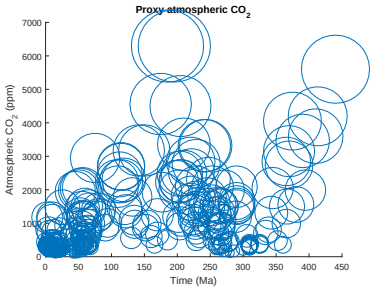

Figure 3.8: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

```
» a=NaN;
» a==NaN
ans =
   0
```

which ... is odd. Having assigned a `NaN` to variable `a` (first line), **MATLAB** is apparently telling you that the value of `a` is not equivalent to `NaN`. Really unhelpful. In fairness, `a` does not have a value and is Not A Number and hence **MATLAB** cannot determine whether or not it is equal to another Not A Number. Better would have been for **MATLAB** to give you an error ... still, it is what it is.

To try and make amends, **MATLAB** provides a *function* to determine whether or not something is Not A Number – `isnana`. This returns *true* (`1`) if the passed variable is Not A Number, and *false* (`0`) if not, e.g.

`isnan`
'`isnan(A)` returns an array the same size as `A` containing logical `1` (true) where the elements of `A` are `NaN`s and logical `0` (false) where they are not.'
Meaning that you can pass any dimension of array (e.g. vector or matrix or 3D), not just a scalar (a single value or 1×1 matrix).

```
» isnan(a)
ans =
   1
```

(because we set `a` to `NaN` earlier), whereas:

```
» isnan(99)
ans =
   0
```

because `99` is a number (integer) and not a Not A Number.

3.3.2 *Other data filtering*

In the example of the observational Riverside temperature data (the data file: `temperature_riverside.txt`), if would be nice to also be able to use `find` ... which can determine all the locations at which a `NaN` occurs[34], by e.g.:

[34] Using `isnan`.

```
» find(isnan(temperature_riverside))
```

... but it is not obvious what to 'do' with the resulting list of *linear indices* for each cell containing a missing value (`NaN`). For instance, you cannot remove a single cell from a $a \times b$ array because an array must contain the same number of elements in each row and the same number of elements in each column.

What you need, is a way of automatically removing each and every row in which one (or more) `NaN`s appear. There are two obvious approaches:

1. use a loop

We could go through the array, row by row, and for every row in which a `NaN` occurs, remove that row. The code (in a new *script* file)

could look something like the following and which works ... but maybe is a little complicated and/or contorted:

```
%%% script to load data, remove NaN-containing rows
% load dataset
data=load('temperature_riverside.txt');
% determine number of rows of data
n_max = length(data);
% initialize row count to the first row
n = 1;
% loop through all rows
while (n <= n_max),
   % find all (any) NaNs in current row
   found_nans = find(isnan(data(n,:)));
   % did you find any NaNs?
   % the vector found_nans is not empty if you did!
   if (~isempty(found_nans));
      % then remove entire row
      data(n,:)  = [];
      % remember to update total number of rows!!!
      % (there is one less now)
      n_max = n_max - 1;
      % NOTE: don't update row count
   else
      % move on to next row
      n = n+1;
   end
end
```

2. cheat!

(not really)

There is a relatively new **MATLAB** *function* that achieves just this: `rmmissing` (see Box). Much simpler code that does exactly the same job as above, would then look like:

```
%%% script to load data, remove NaN-containing rows
% load dataset
data=load('temperature_riverside.txt');
% remove problem rows!!!
data = rmmissing(data);
```

Now, for any row of data in which a `NaN` occurs, **MATLAB** automatically removes the entire row from the array.

If you prefer to break things up – we need not assign the row-removed array back into the same variable and could create a distinct new one, e.g.

```
% remove problem rows (and assign to a new variable)
data_new = rmmissing(data);
```

`rmmissing` – 'Remove rows or columns with missing entries'.

In the simplest usage:

```
B = rmmissing(A);
```

Removes rows containing missing data elements from array `A`, assigning the results to array `B`.

MATLAB defines missing data as:

- `NaN` - for number arrays
- `<missing>` - for string arrays
- blank character `[' ']` - for character arrays
- empty character `''` - for cell arrays

(see **help** for further information and examples)

3.3.3 *Some miscellaneous and useful data manipulations techniques*

Sometimes you will find you need data as a *vector*, but you only have it in the form of an *array*. **MATLAB** provides the function `reshape`[35] for the express purpose of re-configuring the shape of an *array*, such as turning a *matrix* into a *vector*, or *vice versa*.

For instance, given a $3x3$ matrix `D`:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

how do we turn this into a $9x1$ column vector, i.e.

$$\begin{pmatrix} 1 \\ 4 \\ 7 \\ 2 \\ 5 \\ 8 \\ 3 \\ 6 \\ 9 \end{pmatrix}$$

You can use `reshape` in 2 different ways:

1. Firstly, you can explicitly specify the new array shape you want.[36] e.g.

```
Dvector = reshape(D,[9,1]);
```

2. Alternatively, if you know you want a single column vector and cannot be bothered to work out how many rows you need, **MATLAB** will kindly pick up the slack via a slightly different usage of `reshape`:

```
Dvector = reshape(D,[],1);
```

Here you are specifying one column, but 'whatever' (`[]`) rows.

(Having created a vector containing all the numbers, you can now find the standard deviation: `std(Dvector)`.)

Obviously, if you want a row, rather than a column vector – either transpose the column vector to row vector shape, or specify the format of a row vector in the first place when using `reshape`:

```
Dvector = reshape(D,[1,9]);
```

[35] See `help` and Box

`reshape`

Use `reshape` to transform data in an array of one shape (i.e. configurations of rows and columns), into another. **MATLAB** `help` is OK on this and for the main usage of the *function*, says:

'B = reshape(A,sz) reshapes A using the size vector, sz, to define size(B). For example, reshape(A,[2,3]) reshapes A into a 2-by-3 matrix.'

In this usage you need to specify the rows and columns of the resulting array.

NOTE that the array you turn it into to, can have a single row, or a single column (and hence be a vector), but you need to specify this with a `1`.

Also note that the total number of elements in the array must be conserved, so if you turn an $n \times m$ array into a $p \times p$ array, then it must be true that:

$n \times m = o \times p$ There is also a convenient second usage, that will attempt to automatically determine the row or columns needed to make $n \times m = o \times p$ true, given either o or p. For example:

```
B = reshape(A,2,[])
```

in the previous example will automatically determine that 3 columns are needed. Conversely,

```
B = reshape(A,[],3)
```

will determine that 2 rows are required to meet the $n \times m = o \times p$ criteria.

This usage is particularly convenient for making vectors, e.g.:

```
B = reshape(A,[],1)
```

[36] Obviously, the total number of elements in the array must be conserved.

126 str='do you like bananas?';

3.3.4 *Data interpolation*

Interpolation? What is it and why would you do it? We'll answer this via an example.

First download the ice-core dataset of atmospheric CO2 over the past 800,000 years, recovered from the Dome C site on Antarctica – filename: icecore_co2.txt on the course webpage. Load it in and assign the the variable `co2`. Start by plotting it (your choice of **MATLAB** plotting function) to see what you are dealing with.[37]

[37] First column / x-axis values are age, in years, and 2nd column / y-axis values are CO2 concentration, in units of ppm.

So what if we wanted to know the average (mean) value of atmospheric CO_2 over the last full glacial cycle, i.e. between now (age zero) and the end of the previous interglacial period, about 115,000 years ago.

So firstly, you might use your most excellent **MATLAB** skills to extract all the data corresponding to this specific interval – i.e. all the ages (and corresponding CO_2 values, between zero and 115,000 years, or rather, less than or equal to 115,000 years. You should know this requires the find function, and that the range of indices if given by (assuming the data array you loaded in is called `co2`):

```
» a=find(co2(:,1)<=115000)
```

which simply says: take all the elements in the 1st column of the array `co2` (`co2(:,1)`), and find the indices of all the elements with a value equal or less than 115,000. To select just the first 115000 years of data in `co2` is then just a matter of:

```
» co22=co2(a,:)
```

and check that this does indeed give you the correct portion of data and has assigned it to the array `co22`. (Maybe plot to confirm.)

It is worth pausing at this point – this is a common, and powerful, usage of `find`, and of *indexing*, and you should be sure you understand it before moving on. What this line is saying is: take both columns of the array `co2` – select all the elements (rows) defined by the vector `a`, and assign the result to `co22`.

OK, so we are progressing well towards answering the question – the mean CO_2 value over the last glacial cycle (last 115,000 years). In fact – try answering that now (using `mean`). You should end up with a value of $245 ppm$.[38] The question is – do you 'believe' it? Look at the plot – do you think that value is representative of the average?

[38] Note that **MATLAB** will report the value in a scientific notation with a power (here 10^2).

To make the problem more obvious – repeat the above exercise, but now consider only the past 40,000 years. From the plot, high, interglacial CO_2 values characterise only the last 10,000 years or so, with a transition over 5,000 years or so before that. From 15,000 years and back to 40,000, CO_2 is clearly bumping along its lowest values. What would you guess the mean CO_2 value is? Now try it. I get

$249 ppm\ CO_2$. Does that look correct to you, across the past 40,000 years?

If you were previously using `plot` to plot the data, now try `scatter`. It should be much more obvious what is going on now – you have very uneven data sampling in time – the bulk of the data is from the last 10,000 years or so, and there are very few data points older than about 22,000 years. When **MATLAB** calculates a mean, it is of the data points, and an uneven data sampling will give a biased, unrepresentative value.

We need to interpolate the data – place it on an evenly sampled-in-time basis. The **MATLAB** function to interpolate vector (1D) data is `interp1` (see `help`/Box).

The first thing we need, to use `interp1`, is a vector of points in time, that we are going to interpolate our data on to. As a rule, the vector should ideally not extend in value beyond the minimum and maximum values of the original axis, but we'll ignore this for now. We might pick ... 1,000 years as a simple sampling interval, and so to create this new axis vector, we would write:

```
» xi=[0:1000:40000];
```

assuming we stick with the 0-40,000 year interval. The `interp1` function requires that you pass this vector, along with the original time (x-axis) vector, and the original data (y-axis) vector, and will give you a new data vector, with values corresponding to the time points defined by `xi`. Like this:

```
» yi=interp1(co22(:,1),co22(:,2),xi);
```

If you prefer to break things down[39] so that the process is a little clearer, maybe first extract from the original data array, an x-axis (time) *vector*:

```
xold=co22(:,1);
```

and then extract a y-axis (data) *vector*:

```
yold=co22(:,2);
```

and then do the interpolation:

```
yi=interp1(xold,yold,xi);
```

Either way, now scatter-plotting the interpolated data:

```
» scatter(xi,yi);
```

should result in an obviously evenly-spaced data plot.

`interp1`

```
yi = interp1(x,y,xi)
```

will interpolate the y-axis values located at x-axis points given by the vector `x`, onto the x-axis points given by vector `xi`. The resulting interpolated y-values are assigned back to `yi`.

By default the interpolation methods used is linear. For a different interpolation method, use the varient of the function:

```
yi =
interp1(x,y,xi,method)
```

where method is one of: `'nearest'`, `'linear'`, `'spline'`, `'cubic'` ... (for a fuller list, see `help`).

To extrapolate outside of the domain spanned by the (original) x-axis vector `x`, specify:

```
yi =
interp1(x,y,xi,method,'extrap')
```

[39] And then you might also make the variable names REALLY explicit, and have `xold`, `yold`, `xnew`, `ynew` or something.

[**OPTIONAL**] We could now use the *function* `mean`, except ... if you were paying attention, we extrapolated outside of the range of the extracted data into the array `yi`. But you know how to handle this situation, i.e. `find`-ing and removing the offending `NaN` rows, or use `nanmean` if you have access to the required **MATLAB** toolbox.

Or, you could re-do the interpolation, but interpolate from the full, original data array, which you know extends way past 40,000 years. And ... specify the very first time point as 1,000 years rather than zero. e.g.

```
x1=[1000:1000:40000];
y1=interp1(co2(:,1),co2(:,2),x1);
```

Well ... it doesn't work, which is sort of pretty 'real world' problemesk. The issue is that there is a duplicate year – i.e. 2 CO_2 values with the same year.[40] How to find them? Well, you saw earlier the function `mode`, which return the most popular value in an array. If we do:

[40] The **MATLAB** interpolation function requires a strictly monotonically increasing (with no duplicates) old and new x-axis vector.

```
» mode(co2(:,1))
ans =
   409383
```

Ha ha, so the year 409,383 is duplicated.[41] How to find this ...

[41] The absence of duplicated year values, or rather, there only being one of each individual number, would result in **MATLAB** returning the very first value in the vector. The only confusion here would be if the very first value was itself duplicated ...

```
» find(co2(:,1)==409383);
```

or if you prefer (and neater):

```
» » rows=find(co2(:,1)==mode(co2(:,1)))
ans =
   531
   532
```

giving us the row numbers. We could be cleaver, and create a single entry for this year, with the CO_2 value formed of the mean of the duplicate entries:

```
» co2mean=mean(co2(rows,2));
```

... replacing the first CO_2 value ...

```
» co2(rows(1),2)=co2mean;
```

... and then delete the second.

```
» co2(rows(2),:)=[];
```

Phew! Now try the interpolation again. Plot (`scatter` and/or `plot`). Find the mean CO_2 value over 0-40,000 (technically, now we have restricted the data to 1,000-40,000). Does this seem more reasonable? Repeat for 0-115,000 years (or 1,000-115,000 years). Also

try out carrying out an interpolation with closer spacing, say 500 or 100 years.

Finally, if you read the (Box or `help`) details about the function `interp1`, the more recent **MATLAB** version releases enable you to extrapolate outside of the data domain. So instead of having to restrict the `xi` vector to starting at year 1,000, you can start at year zero:

```
» x1=[0:1000:40000];
» y1=interp1(co2(:,1),co2(:,2),x,'linear','extrap');
```

 str='do you like bananas?';

3.3.5 *Data (row) deletion*

For some practice in some 'real world' data filtering and a little further data manipulation and graphing: download the historical temperature dataset for Riverside[42]:

[42] NOAA

temperature_riverside.txt

If you view the data file in a text editor or import into **Excel**, you can read the header information and find out what the different columns of data are (I am not telling you!). Note that the (1st) line of the file containing the column header labels starts with a % symbol (I added this to tell **MATLAB** to ignore this line and not attempt to read in the 'data' on it). (I also ensured that any missing data was indicated with a `'NaN'` to make the `load` go more smoothly.)

Load in this file (and you might assign the read-in data to a different and shorter variable name):

```
» tdata = load('temperature_riverside.txt');
```

and maybe take a look at what you have by opening `tdata` in the variable window.

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

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

or, remembering that end is shorthand for the last element or index number, you could also write:

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

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

You could certainly simply plot both global mean and Riverside annual mean temperatures alongside each other, using the same units, e.g. °F as you have previously converted the global mean temperatures to °F, which is the same units as the Riverside temperature data. You could have 2 sperate plots and visually compare them, but this is not very cleaver nor necessarily useful in making any sort of quantitative comparison. For instance – contrast the global data (rescaled to absolute degrees F) in Figure 1.10, with the Riverside data, in Figure 3.9.[43]

`comment symbol`

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

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

[43] There are also sone odd-looking artifacts ('spikes') in the raw data that we will want to deal with in some way.

There are two main problems [44] in making a useful comparison – firstly, the two data sets are on different y-axis scales (but luckily on the same x-axis, year scale), with the global data temperature scale going from 56 to 59°F, and the local, Riverside temperature data scale going from 50 to 85°F.

[44] Plus artifacts in the raw data.

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

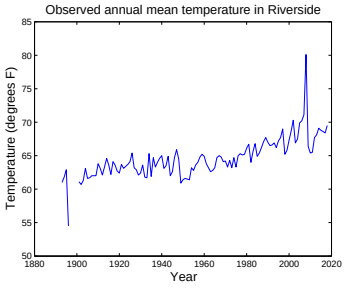

Figure 3.9: Observed annual mean surface temperature in Riverside.

Recall that the original global temperature data was given as an anomaly compared to the average over some baseline (or reference) period – in this case, year 1910 to 2000. If we treated both data sets the same, and transformed the Riverside temperature data into a comparable anomaly, a more direct comparison could be made.

To create an anomaly of the Riverside temperature data, relative to the mean of the data for years 1910 to 2000, requires that we know what the mean of the data is over the years 1910 to 2000. Using the **MATLAB** function `mean`, you should be able to work out that you need:

```
» tdata_mean = mean(tdata(:,end));
```

and then all you need to is is to subtract that from the annual data, e.g.

```
» mean(tdata(:,end)) = mean(tdata(:,end))-tdata_mean;
```

(which is talking the last column of the array, subtracting `tdata_mean` from all the values, and writing that vector back to the last column. EXCEPT, the answer is apparently ...

```
» tdata_mean = mean(tdata(:,end))
tdata_mean =
NaN
```

`NaN`??? Oh no ...

Actually, more recent versions of MATLAB allow you to ignore `NaN`s in basic stats functions such as `mean`. For example, referring to **MATLAB** `help` and re-trying the example above:

```
» tdata_mean = mean(tdata(:,end),'omitnan');
```

tells **MATLAB** to calculate the mean while ignoring any `NaN`s in the *vector*.

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

One could address this by removing the years with (substantially) incomplete monthly data from the data file[45] and prior to loading into **MATLAB**. Or would could process the data once in **MATLAB**. This can be done by assigning to particular row (vector) of data, an empty vector ([]).

Taking first a simple example of a column vector:

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

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

```
» A = [1; 2; 3];
```

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

```
» A(2) = [];
```

Similarly, to remove the 2nd row of:

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

we could type:

```
» B = [1, 4; 2, 5; 3, 6];
» B(2,:)  = [];
```

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

So back to the temperature data – to remove the row containing the year 2008 data for example[46], which is row 11, we would write:

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

Play this 'game' – deleting as many row as you think result in biased means (because of missing monthly data)[47], with the Riverside

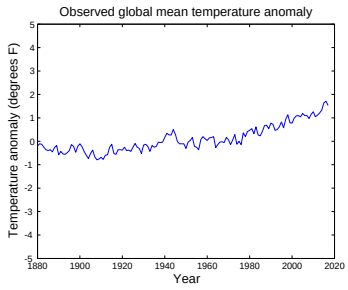

Figure 3.10: Observed global annual mean surface temperature anomaly, relative to the mean of 1910 through 2000.

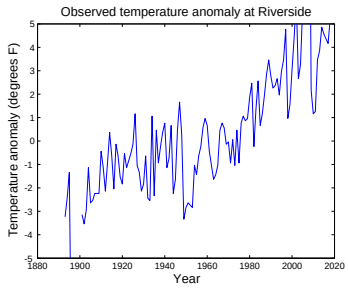

Figure 3.11: Observed annual mean surface temperature anomaly, relative to the mean of 1910 through 2000, at Riverside.

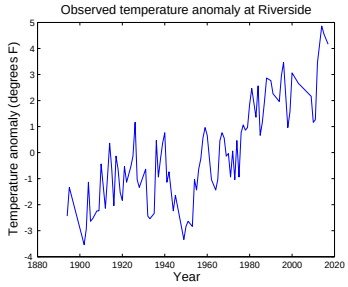

Figure 3.12: Observed annual mean surface temperature anomaly, relative to the mean of 1910 through 2000, at Riverside, filtered to remove years with missing monthly data.

[45] i.e. simply deleting the line in the file.

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

[47] (being aware that as you delete rows, the numbering of the subsequent rows changes as the array size shrinks ...)

temperature data, and re-plot the results. For example, the result of removing ALL the rows with missing monthly data[48], results in Figure 3.12.

[48] (There are simple and quick ways of doing this in **MATLAB** that we will see later.)

Finally, go back to the original global mean anomaly values (re-load the data set if necessary) and convert from the anomaly in °C to °F (i.e. simply multiply by 1.8 – no offset (32°F) is required in this particular case). If you additionally, chose and set a sensible common y-axis scale for both plots, you might end up with a pair of graphs looking like Figures 3.10 and 3.11. [49]

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

```
134  str='do you like bananas?';
```

3.4 *Even nicer graphing and graphics*

There are a bunch of simple **MATLAB** drawing and text placement commands that can help improve the look and feel of scientific plots, or even replace the provided plotting functions (i.e. you can create your own bespoke plotting functions). There are also a variety of options for altering the axes, axes tick-marks, axes tick-mark labels, etc that can be useful.

3.4.1 *Drawing lines (and using handles)*

We'll start with some simple line drawing.

At the command line – open a new figure window (`» figure;`). Then before anything else, do a `» hold on;`.

When **MATLAB** draws lines and shapes and places text, it needs to know the coordinates of where to place things. It is not possible (I think) to draw directly in the figure window – **MATLAB** needs what is known as a *frame* to put things in, and the easiest way to do this is to create a set of axes. Having opened a new figure window (even though you have not plotted any data!), set the x- and y-axis range[50]:

```
» axis([0 100 0 100]);
```

The resulting figure is really not very exciting (Figure 3.13).

There are 2 ways to draw a line (examples assuming the previous Figure window remains open with `hold on` ...):

1. `plot`
   Are you recall, `plot` will plot a sequence of (x,y) points, and by default, join the points up with a line. If we wanted a diagonal line, from the origin to the mid-point of the plot area, we could invent a pair of vectors to define the two points we need – at (0,0) and (50,50):

   ```
   X1 = [0 50];
   Y1 = [0 50];
   ```

   and then plot the resulting points as a `plot` plot:

   ```
   plot(X1,Y1);
   ```

   You should now see something like Figure 3.14. (If you find that the plot area has been re-sized such that the x- and y-axes now both go from 0-50, then you have forgotten to do a `hold on`.)

2. `line`
   MATLAB provides a specific command for drawing lines ... `line`. In its simplest usage, it is a little like `plot`, except taking only a single pair of x- and y-coordinate values.
   To use `line` to draw a 2nd line segment, starting at (50,50) and terminating at (100,0), we create another pair of vectors to define these points:

   ```
   X2 = [50 100];
   Y2 = [50 0];
   ```

   where `X2` contains the 2 x-coordinate values, and `Y2` contains the 2 y-coordinate values, such that you are plotting a line between (`X2(1)`,`Y2(1)`) and (`X2(2)`,`Y2(2)`).

`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.

You can also specify line colors and styles in a similar way to when using `plot`, e.g.

```
» line ...
([x1 x2],[y1 y2], ...
'Color','red', ...
'LineStyle','-')
```

plots a red dashed line.

[50] Here taking the example range of 0-100 on both axes.

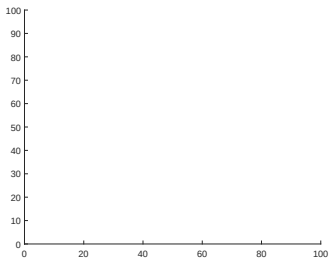

Figure 3.13: Figure window with axes.

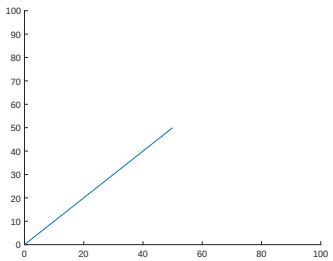

Figure 3.14: Figure window with single line segment (via `plot`).

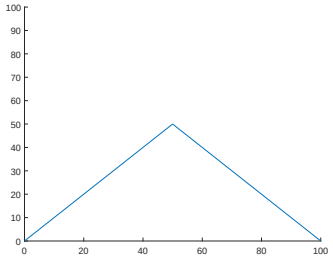

Figure 3.15: Figure window with a second line segment (via `line`).

To draw it:

```
line(X2,Y2);
```

as shown in Figure 3.15.

Obviously, both line segments could equally well be drawn using `plot` or `line`.

If you are just drawing, rather than annotating a plot with axes, then you might want to turn off, or 'hide', the axes tick marks and tick labels. To do this, we first need to find the special **MATLAB** ID of the axes, which helpfully, because you only have one set of axes and have just been using them, is the 'current axis'. To do this, we use the `gca` function, which returns the *handle* (**ID**) of the axes:

```
» h = gca;
```

Having got the axis *handle*[51], we can now 'set' the properties of the axes, using `set`:

```
» set(h,'XTick',[],'XTickLabel',"");
```

What this does is to tell **MATLAB**: take the graphics object with the **ID** contained in variable `h` (which we just retried via the `gca` function), and set (which is why we use the command `set` ...) the following properties:

- `'XTick',[]` – set the number and position of tick marks on the x-axis, to the contents of the empty vector `[]`.
- `'XTickLabel',"` – set the labels applied to the tick-marks, to `"` (i.e. no text).

Actually, in this example, the 2nd graphics parameter set (the labels) is sort of redundant, as there are no tick-marks in the first place ...

To see how different combinations of settings pan out, try:

```
» set(h,'XTick',[0 50 100],'XTickLabel',");
```

(3 small inwards ticks, no labels)

```
» set(h,'XTick',[0 50 100],'XTickLabel','cat');
```

:o)
and/or:

```
» set(h,'XTick',[0 50 100],'XTickLabel',{'cat',
'dog', 'rabbit'});
```

where `{'cat', 'dog', 'rabbit'}` is actually a 3 string *cell array* (the curly brackets are important to the syntax).

All this insanity should be looking like Figure 3.16 (if we also remove the y-axis ticks and labels[52]).

---

[51] It is worth omitting the `;` in order to see the properties associated with the axes, and in fact, it is worth clicking on `Show all properties` to see a list of everything that can be edited and adjusted.

ALL these can be changed if you ever want!!!

[52] It is sufficient just to type:

```
» set(h,'YTick',[]);
```

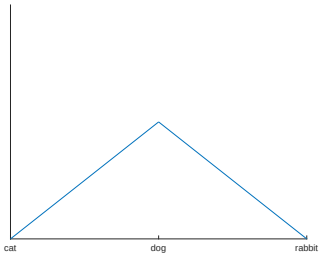

Figure 3.16: (no comment).

An alternative way to create a figure to draw on, without having to remove the axes ticks and labels etc etc, is to create the axes as 'invisible' in the first place. To try this – first create a new figure window.[53]

```
» h = axes('Position',[0 0 1 1],'Visible','off');
```

Here, `'Position',[0 0 1 1]` specifies that the axes area should fill the window, and `'Visible','off'` says to make the axes invisible. (We keep a copy of the *handle* variable, `h`, just in case we need it later.) To set the axes scales to the same 0-100 limits as before:

```
» axis([0 100 0 100]);
```

Then `hold on` to start drawing things![54]

Try drawing some lines in this window (remembering the 0-100 axes limits when making up (x,y) co-ordinates).

The command `set` can be used in the context of any (I think?) graphics object, i.e. any individual component part of a final plot such as the axis line itself, the axis ticks, plotted lines and points, etc. For example, in creating the line segment previously:

```
» h = line(X2,Y2);
```

you could store a copy of the *handle* of that line segment – here it is being assigned to the *variable* `h`. With this, you can now change the properties of the line (after you have drawn it). such as by:

- `» set(h,'LineWidth',2.0);`
  will change the line width to 2.0 (points).
- `» set(h,'Color',[1 0 0]);`
  will turn the line red, using the *RGB* (red-green-blue) notation: `[1 0 0]`[55].
- `» set(h,'LineStyle',':');`
  will make the line dotted.

Note that here, you are setting line properties after you have created the graphics, whereas earlier in eg. using `plot` to graph data, you specified the properties at the same time as you draw the lines. Both ways are valid, but being able to change properties later and after plotting, gives you greater flexibility. Note that after plotting a graphic, you can also edit and adjust its properties in the Figure window itself (via the **GUI**)

Using a *handle* also now enables you to complete an earlier plot. For the proxy CO_2 data where you color-coded the points and added a `colorbar`, there was not actually any indication of what the color

[53] If you find yourself drowning in figure windows, remember that `close` closes the current window, and `close all` closes all of them.

[54] For some reason ... you need to do `hold on` only after creating axes frame ...

[55] Alternatively:

```
set(h,'Color','r');
```

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

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

The revised plot should then end up looking something like Figure 3.17 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.

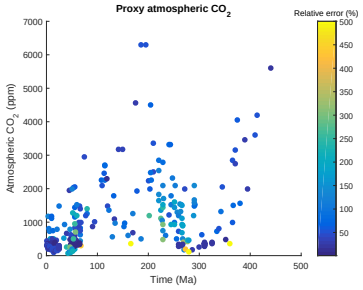

Figure 3.17: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

An obvious use for drawing lines on plots, is to annotate them. e.g. placing a text label (we'll see shortly), with a line pointing from the text to a specific feature. You can do with with a simple line and hence the `line` command.

It would be more handy and in fact common, to include an arrow head to make clear that the line is pointing to something. This can in theory be done by drawing 2 more, shorter lines, but is no fun at all[56]. **MATLAB** hence provides the function `quiver`. `quiver` is commonly used for plotting fields of arrows, but can equally be used to create a single arrow – much like earlier you used `plot` to draw just a single pair of joined up points and hence a line. However, rather than take a pair of (x,y) points – one for the start and one for the end, of the line, `quiver` takes an (x,y) location for the start of the arrow, and then the length in the x and y directions.

Consistent with the previous example, we were starting the line segment at (0,0), and then extending the line to (50,50). The length vector in this case is also [50 50]. So, given the specific syntax and input parameter format required by **MATLAB** for this function[57,58]:

```
» quiver(0,0,50,50,0);
```

To make the syntax clearer and that we are passing 4 vectors (of x and y origin locations, and x- and y-axis lengths), we could also write:

```
» quiver([0],[0],[50],[50],0);
```

[56] True fact – I have tried it :(

[57] Here – the last, 5th parameter (`0`), tells **MATLAB** not to auto-scale the arrow.

[58] If your arrow head is hard to make out – try creating a new figure window. You can also use `cls` to clear all the graphics in the window (i.e. and not have to re-generate the figure window.

3.4.2 *Colors*

You are already familiar with setting colors for lines, with the notation: `'r'`, or `'b'` (for red, blue, respectively). This is nice and simple and so totally fabulous ... except there are a limited number of colors available in this notation (see Box).

Hence there is an alternative that enables a more exact specification of color. In this particular scheme – red-green-blue, abbreviated to *RGB*, you set the intensity of red, green, and blue, on a scale of 0 to 1. And supply this in a vector format to **MATLAB**. For example:

- `[0 0 0]` – zero intensity of all of R, G, B => black.
- `[1 1 1]` – 100% intensity of all of R, G, B => white.
- `[1 0 0]` – 100% intensity R, none for G and B => red.
- `[0.5 0.5 0.5]` – 50% intensity of all of R, G, B => grey.
- `[0.5 1.0 0.5]` – light green.

Play around with some RGB value combinations, plotting shapes, or filled circles, e.g.

```
» scatter(50,50,1000,[0.25 0.75 0.25],'filled');
```

which scatter-plots a single point at location (50,50), it makes the default circle size `1000` points (3rd parameter), specifies a particular color (`[0.25 0.75 0.25]`), and then demands that the point is 'filled'.

A rendition of the RGB color scale is shown for reference in Figure 3.18.

MATLAB quick colors:

- `y` – yellow
- `m` – magenta
- `c` – cyan
- `r` – red
- `g` – green
- `b` – blue
- `w` – white
- `k` – black

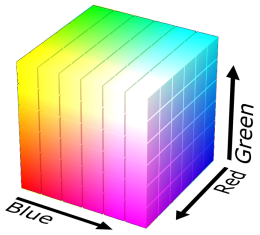

Figure 3.18: RGB scale. By SharkD - Own work, GFDL, https://commons.wikimedia.org/w/index.php?curid=3375025

3.4.3 *Shapes*

For 2D shapes – you can draw polygons using the *function* `patch`. This takes as parameter input, a vector of x-coordinate positions, then a vector of y-coordinate positions, and as a 3rd parameter, the color for the object.

So in our previous example, with the x- and y-axes going from 0-100, say we want to draw a square in the middle, 20 units on each side. We could create our vector of x-axis coordinates as such:

```
» X1 = [40 40 60 60];
```

and for the y-axis ... (some care is needed and often it might be helpful to sketch out the coordinate pairs and positions on paper):

```
» Y1 = [40 60 60 40];
```

In this `(X1,Y1)` point notation, we are telling **MATLAB** to plot a shape with vertices at: $(40,40), (40,60), (60,60), (60,40)$

`patch`

To plot one (or more) filled polygonal regions, pass 2 vectors of the x and y as the coordinates for each vertex, e.g.

```
» patch(X,Y,C)
```

where `X` and `Y` are the x- and y-coordinates of the vertices. `C` determines the polygon's color.

For example, a red square would be:

```
x = [0 1 1 0];
y = [0 0 1 1];
patch(x,y,'red')
```

To draw the shape, simply pass the 2 coordinate vectors to `patch`, along with any line style specification:

```
» patch(X1,Y1,'r');
```

as shown in Figure 3.19.

Note that the order of the (x,y) pairs, matters, as **MATLAB** draws the line segments between the vertices in the order that they are given to **MATLAB** (as per for `plot`). For the same (x,y) pairs, try creating `X1` and `Y1` vectors with the pairs in a different order and see what happens, e.g. change the x-vector to:

```
» X1 = [60 40 60 40];
```

Figure 3.19: Square.

and see what happens when you call `patch` again.

Slightly changing the values of the (x,y) pairs can also give you a diamond (-ish):

```
» X2 = [40 50 60 50];
» Y2 = [50 60 50 40];
» patch(X2,Y2,'c');
```

Figure 3.20: Alt square.

as shown in Figure 3.20.

`patch` is in fact much more flexible than I have shown so far, and in fact, will draw any polygon. Consider this sort of slightly random series of x and y coordinates:

```
» X3 = [20 40 60 80 60 40];
» Y3 = [50 60 50 60 40 30];

» patch(X3,Y3,'g');
```

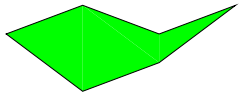

Figure 3.21: Random polygon.

gives Figure 3.21.

Try designing/playing about with different shapes. Perhaps sketch them out on paper first and list down the coordinates before telling **MATLAB**.

If you have the **MATLAB** Image Processing Toolbox installed, then you can use the command `viscircles` to draw circles.

If not – a crude but sometimes effective alternative, is to `scatter` plot a single point ((x,y) location), and set a large size value for the circle. For example:

```
» scatter(50,50,1000);
```

or filled:

```
» scatter(50,50,1000,'filled');
```

3.4.4 *Placing and making text 'nice'*

There is not much to placing text and specifying its properties. The **MATLAB** command for writing a string to a figure window, is `text`. That's it! (see Boxes)

For instance, you could write:

```
» text(25,25,'bananas');
```

and the text `bananas` will appear at location $(25, 25)$ on your plot.

Additional parameters can be added to change font, size, etc (see Box), e.g.:

```
» text(25,25,'bananas','FontSize',24,'Color',[0 1
1]);
```

for big light blue 'bananas'.

`text`

```
text(X,Y,STRING);
```

will write the string contained in the variable STRING (or you can pass the text as a string directly), at location (X,Y).

Note that by default, MATLAB alights the left-hand edge of the text with the X coordinate position, and the mid-point of the string vertically, with the Y coordinate. i.e. the string is *left-aligned* and *centered* vertically.

A variety of additional properties can be set at the time, e.g.

```
text ...
(X,Y,STRING,'FontSize',12);
```

specifies a 12 pt font size. Other common parameter options include:

- `'FontName'`
- `'Color'`
- `'Rotation'`
- `'HorizontalAlignment'`
- `'VerticalAlignment'`

See **MATLAB** `help` for more details.

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 {}.

3.4.5 *Creating color maps*

As mentioned earlier – **MATLAB** enables a range of different color scales (*colormaps*) to be used in (esp. contour) plotting and provides around dozen built-in possibilities (see Box).

Taking the earlier example of loading and plotting the global topography data:

```
» data = load('etopo1deg.dat','-ascii');
» imagesc(data);
```

gives Figure 3.22, and

```
» colormap('hot');
» imagesc(data);
```

gives Figure 3.22.

You can also define your own *colormap*. *Colormaps* are simply a matrix of $[RGB]$ colors. The most trivial *colormap* would be:

```
» cmap1 = [0 0 0];
» colormap(cmap1);
```

creates and applies a *colormap* containing a single color (black). Try it ... but ti is clearly not very useful ...

Better, would be:

```
» cmap2 = [0 0 0; 1 1 1];
» colormap(cmap2);
```

which creates and applies a color scale containing 2 colors - black and white and when used for the topography data, gives Figure 3.24. [59]

You can keep adding colors, e.g.

```
» cmap3 = [0 0 0; 0.5 0.5 0.0; 0.0 0.5 0.5; 1 1 1];
```

but this is a lot of effort to keep adding single additional colors. What you really want to do, is to define end-member colors, and then tell **MATLAB** to *interpolate* in between these. Recalling back a couple of subsections:

```
» ynew = interp1(xold,yold,xnew);
```

takes the y-values (`yold`) at x-values `xold`, and interpolates onto the x-values defined by the vector `xnew` (and assigns the new y-values to vector `ynew`). For instance, if we have the following crudely spaced data[60]:

`colormap` (2)

As mentioned earlier, **MATLAB** has a number of 'colormaps' built in, which are:

- parula (default)
- jet (old default ... avoid ...!)
- hsv
- hot
- cool
- spring
- summer
- autumn
- winter
- gray
- bone
- copper
- pink
- lines
- colorcube
- prism
- flag

and which you can set by:

```
» colormap NAME
```

(or `colormap(NAME)`), e.g.:

```
» colormap 'hot'
```

(or `colormap('hot')`)

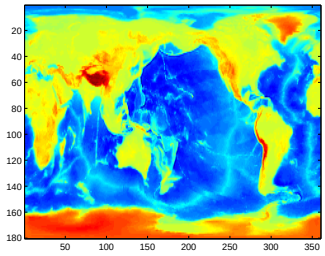

Figure 3.22: Global topography plotted with the default **MALTAB** color scheme.

[59] Remember, `imagesc` plots using the maximum number of colors available, and in this example, the mid value between the deepest place in the ocean and highest point on land, divides the colors into black and white – within specifying a particular scale, this color separation does not occur at zero (sealevel)

[60] In MATLAB notation:

```
xold = [0 3 7 13 16 22
30];
yold = [0.2 0.6 0.7 0.3
0.3 0.1 0.0];
```

$$\begin{pmatrix} 0 & 0.2 \\ 3 & 0.6 \\ 7 & 0.7 \\ 13 & 0.3 \\ 16 & 0.3 \\ 22 & 0.1 \\ 30 & 0.0 \end{pmatrix}$$

and we wanted to create an interpolated dataset from 0.0 to 30.0 (in x-axis values) in steps of 1.0, we would first create the new x-axis vector that the data will be interpolated on to:

```
xnew = [0.0:1.0:30.0];
```

and then we would write:

```
» ynew = interp1(xold,yold,xnew,'spline');
```

and obtain the interpolated data as shown in Figure 3.25. (Try this.)

We can do something similar for the *colormaps*. Consider the simple end-member black-to-white white scale:

```
» cmap2 = [0 0 0; 1 1 1];
```

We can write this as points along a vector x (the axis not representing anything in particular – the number of the color, or simply the normalized distance between the extreme end-member colors), together 3 color vectors (for the separate red, green, and blue component values):

$$xold = \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix}, rold = \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix}, gold = \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix}, bold = \begin{pmatrix} 0.0 \\ 1.0 \end{pmatrix}$$

and:

```
» xold = [0.0; 1.0];
» rold = [0; 1];
» gold = [0; 1];
» bold = [0; 1];
```

If we want to create a scale of 11 total (from 0.0 to 1.0 in steps of 0.1) different colors, we can create a new x vector to interpolate on to:

```
xnew = [0.0:0.1:1.0];
```

and then either interpolate the 3 color vectors separately:

```
rnew = interp1(xold,rold,xnew,'spline');
gnew = interp1(xold,gold,xnew,'spline');
bnew = interp1(xold,bold,xnew,'spline');
```

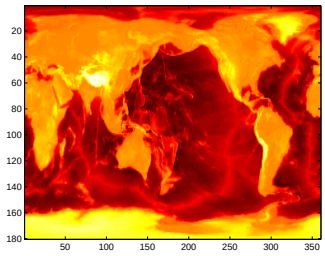

Figure 3.23: Global topography plotted with `hot`.

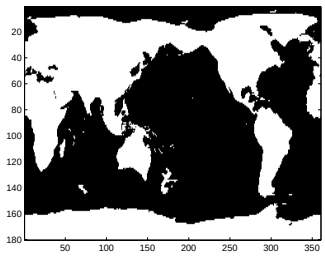

Figure 3.24: Global topography plotted with a basic black+white dual color scheme.

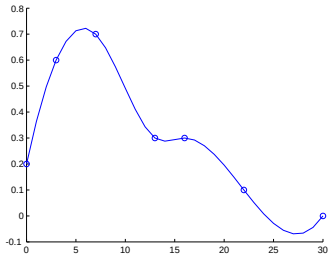

Figure 3.25: Comparison of sparsely sampled data (points) compared with a more finely spaced spline interpolation (solid line). (x-axis and y-axis are both unit-less.)

or **MATLAB** allows us to interpolate all together if we first combine the sperate vectors:

```
mapold = [rold gold bold];
```

and then:

```
mapnew = interp1(xold,mapold,xnew,'spline');
```

If you now set the new *colormap* (» `colormap(mapnew);`) and re-plot the global topography, you should get Figure 3.26.

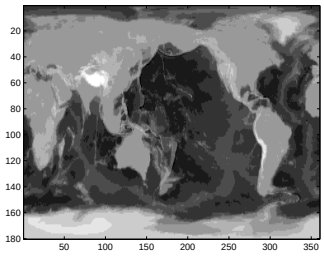

Figure 3.26: Global topography plotted with a user-defined grey-scale.

3.5 Stats (it had to happen ...)

3.5.1 Basic (pretend) 'stats'

We are not going to delve into complex stats here. A variety of stats related functions are included in the **MATLAB Statistics Toolbox**. We'll stick to some simple functions included as standard in the basic package. Useful basic stats-related functions include:

- `min` – the minimum value of ...
- `max` – the maximum value of ...
- `sum` – the sum of a vector of numbers.
- `mean` – the mean of a vector of numbers.
- `std` – the standard deviations of a vector of numbers.
- `var` – the variance of a vector of numbers.
- `median` – the median value of ...
- `mode` – the mode value of ...

For instance, consider vector `A` (integers from 1 to 9, inclusive):

```
» A = [1:9];
```

Try out all of the above functions on the vector. Most of the values you can pretty much guess. The `mode` of the vector is perhaps the only one where it comes up oddly, because the mode of a set of values is defined as the most popular value, yet you have created only one of each value in the vector. So all values are in theory equally the most frequent and **MATLAB** simply returns the first.

So try adjusting the vector, adding an additional '7' at the end:

```
» B = [A 7];
```

Now what is the mode value of the vector `B`?

Sometimes you have have the situation where you have one or more `NaN`s in the data. For example:

```
» C = [A NaN];
```

Now try out some of the same functions on vector `C`. What happens? Why **MATLAB** does this and does not simply ignore `NaN`s, is anypony's guess. I mean, what application could you possibly have where when you ask for the mean of a vector, you are hoping to be told '`NaN`'? There are solutions.

1. Firstly – you could use find, to find and remove `NaN`s from data. So if you have data that includes `NaN`s, you could simply filter them out prior to processing the data. The function for determining whether or not an array element is a `NaN`, is `isnan`)see earlier Box).

2. Or you could create a loop and test each element in turn as to whether or not it is a `NaN` (again, using the `isnan` function).
3. The **MATLAB** functions listed above, all (most?) have an additional optional parameter (see Box) that allows you to direct **MATLAB** to ignore `NaN`s in the data.
4. Lastly, in the **Statistics Toolbox**, there are variants of all (most?) of these functions that automatically ignore `NaN`s, such as `nanmean` (the `NaN`-ignoring variant of `mean`)

Try out each of these solutions, applying them to the vector `C` (or a `NaN`-containing vector of your choice).

All of these functions can also be used on 2D arrays (matrices) ... with care. Consider the matrix:

```
» D = [1 2 3; 4 5 6; 7 8 9];
```

(which has the same elements as `A`, just in a different configuration). What happens when you ask for `mean(D)`? As per help (and the Box): *'If A is a matrix, then mean(A) returns a row vector containing the mean of each column.'* So `mean(D)` is returning the mean of `[1 4 7]`, `[2 5 8]`, and `[3 6 9]`. Try *transposing* the matrix and then using the `mean` *function*. You should see that you now get the mean of the individual rows (rather than columns) of matrix `D`:

```
» mean(D')
ans =
   2 5 8
```

This goes for `sum` and all (most?) the rest of the functions.

If you need the total sum of all the elements in a matrix, or mean of all the elements in a matrix, you can simply *nest* the *functions*:

```
» sum(sum(D))
```

or if you prefer breaking things down into sperate steps:

```
» E = sum(D); » sum(E)
```

However, note that `std(std(D))` is not the standard deviation of all elements in the matrix `D`. Why?

3.5.2 'Real' stats

4
Further ... Programming

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

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

Search algorithms

Lets assume that you have a function:

$y = f(x)$

There are two common cases that you might want to solve (or approximate):

1. The value of x such that the value of $f(x)$ is minimized ($y \simeq 0$).
2. The value of x such that the value of $\frac{dy}{dx}$ is minimized (first derivative $\simeq 0$.

Lets further assume that you can place some initial limits on $x : x_{min} \leq x \leq x_{max}$.

A good place to start in both examples is to test the mid-point of the limits: $f(\frac{x_{min}+x_{max}}{2})$ (In some cases you might instead take the log-weighted mean.)

In case #1 and assuming that $\frac{dy}{dx}$ is positive, if:

$f(\frac{x_{min}+x_{max}}{2}) > 0$

you replace x_{max} with $\frac{x_{min}+x_{max}}{2}$ (the current tested value of x) and if:

$f(\frac{x_{min}+x_{max}}{2}) < 0$

you replace x_{min} with $\frac{x_{min}+x_{max}}{2}$. Keep repeating until the difference y and zero falls beneath some specified tolerance.

In case #2, you need to test the value of $f(x)$ infinitesimally away from $f(\frac{x_{min}+x_{max}}{2})$ to determine whether the gradient is positive or negative (assuming that you do not *a priori* know the derivative function). The idea here is to ensure that the values of x_{min} and x_{max} correspond to positive and negative (or negative and positive) gradients. i.e. $\frac{x_{min}+x_{max}}{2}$ replaces x_{min} or x_{max} according to which has the same sign of gradient.

4.1 *Nested loops*

A helpful device, particularly when dealing with arrays of data in **MATLAB**, is to *nest* loops – i.e placing one loop inside another one. (So far, you have seen single loops, and single loops with *conditionals* inside, but not nested loops.) A generic code for a nested loop might look like:

```
% loop 1 start
for n=1:10
   % loop 2 start
   for m=1:10
      %%% CODE
   end
end
```

Here, the value of `n` cycles ('loops') from 1 to 10 (i.e. the loop goes around 10 times). Then ... for each value of `n`, the value of `m` also cycles from 1 to 10. The code in the middle of the innermost loop is then executed a total of $10 \times 10 = 100$ times.

Try this (in a new *script* file) and confirm that the outer loop cycles around 10 times, and the inner loop ten times for each cycle of the outer loop. (e.g. you might add a `disp` line both within the inner loop, as well as outside of the inner loop (but still within the outer loop), and/or you might add *break points*).

Why would you do this (what use could it be)?

Image you are programming a game of Tic-tac-toe (in fact we will, in a later chapter!). The drawn grid might look like Figure 4.1.[2]

In terms of **MATLAB** and computer programming, we might create a representation of the grid, and assign `0` to unpicked squares, a `1` for where a cross is, and a `2` for where a naught is, as per Figure 5.7 (because we cannot numerically represent an actual cross or circle shape).

To store this information, we could create an *array* in which each location would have a value of `0` (unassigned square, `1` (player #1), or `2` (#player2)), e.g.

$$\begin{pmatrix} 1 & 2 & 0 \\ 1 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

and as per Figure 4.3

OK – ignore the existence of the **MATLAB** *function* `find`, and lets say that you want find the locations of the crosses – '`1`'s in the array code notation. You need to test each and every location in the array (lets call it `tokens`) in turn for whether its contents is a '`1`' or not. We could do this long-hand ...

[2] In this case, player x has obviously already won. What was naughts thinking???

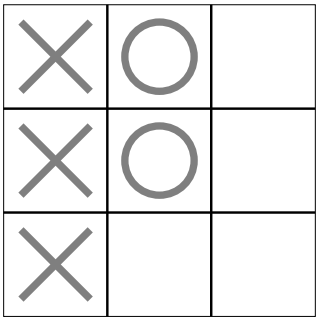

Figure 4.1: Tic-tac-toe game grid.

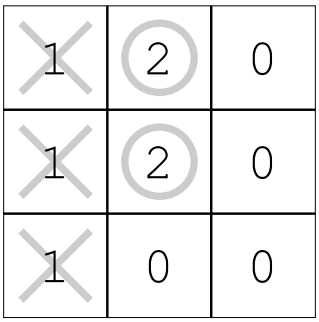

Figure 4.2: Tic-tac-toe game grid with numerical codes overlain.

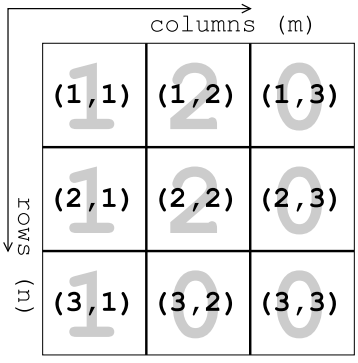

Figure 4.3: Tic-tac-toe game grid – numerical representation.

```
if ( (tokens(1,1) == 1) || (tokens(2,1) == 1) ||
   (tokens(3,1) == 1) || (tokens(1,2) == 1) || ...
```

... but would get desperately tedious pretty quickly.[3] And what if the grid (array) was 100×100? You could have to have $10,000$ tests of an equality in the `if` ...

[3] Note what is being done here – we are using a series of ORs (||) to determine whether any of the array locations (squareso contain a value equal to 1.

The idea then is to *loop* through all the locations in the array in turn. And we do this by: For each row (which we search through in a loop), we loop through all the columns. Our code fragment would then loop like:

```
% loop 1 start
for row=1:3
   % loop 2 start
   for column=1:3
      if (tokens(row,column) == 1),
         %%% CODE
   end
end
```

(where `%%% CODE` is simply a place-holder for some code we might need here to act on having found a player #1 square).

This nested loop ensures that you visit each and every array location in turn – working across every column, for each and every row, as per Figure 4.4 and test for the occurrence of a value of 1.

We could also carry out the search of (*rows, columns*) in the opposite order:

```
% loop 1 start
for column=1:3
   % loop 2 start
   for row=1:3
      if (tokens(row,column) == 1),
         ...
   end
end
```

and now loop through each row (inner loop) for each column (outer loop) as illustrated in Figure 4.5. The result is exactly the same. To some extent, which axis direction you choose as the outer loop is a matter of personal preference.

You can also search in the opposite direction (sign), e.g.

```
% loop 1 start
for column=1:3
   % loop 2 start
   for row=3:-1:1
      if (tokens(row,column) == 1),
         %%% CODE
   end
end
```

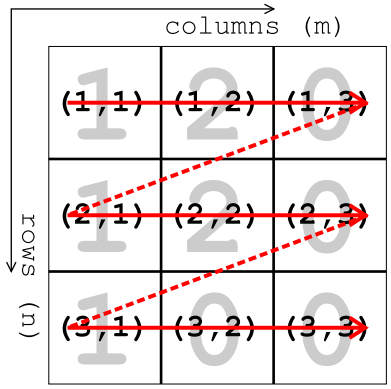

Figure 4.4: Tic-tac-toe game grid – search order: columns then rows.

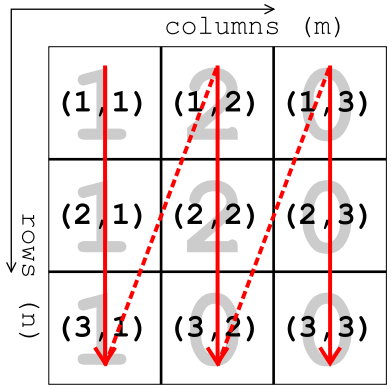

Figure 4.5: Tic-tac-toe game grid – search order: rows then columns.

searches across columns, form left-to-right, but rows in the order bottom-to-top. This perhaps looks a little like how you might visualize a search on a *(lon,lat)* grid(?)

The concept is the same even for very large grids (where you cannot easily draw a graphical representation to help you).

The number of rows and columns do not have to be the same. For example you might want to access information stored in an array that has a cell location for every day of the year. In this case, you might have 12 columns for the 12 months, and 31 rows so that you can accommodate the number of days in the longest month.[4] In fact, in this example, the inner loop – days – might have a different loop maximum, depending on which month, e.g.

[4] **MATLAB** does not allow the number of rows to differ, from column to column – a matrix must have a strictly rectangular shape. **MATLAB** and other programming languages allow the creation of *objects*, that are more flexible.

```
% month loop start
for month=1:12
   % determine length of month
   switch month
      case {1, 3, 5, 7, 8, 10, 12}
         day_max = 31;
      case {4, 6, 9, 11}
         day_max = 30;
      case 2
         day_max = 29;
   end
   % day loop start
   for day=1:day_max
      %%% CODE
   end
end
```

Here, the `switch ... case` structure is used to test for which month (number) it is, and set the day-of-the-month loop limit accordingly. Note the use of curly brackets `{}` in defining the list of elements in the `case` statement. (A single element does not require curly brackets.)

Try this code (in a new *script* file). Add a `disp` line within the day loop to confirm that the correct number of days is being counted up to each month.

Also – try (in the same *script* file, or create a new one) replacing the `switch ... case` structure, with `if ... elseif`. You will need to use OR (`||`), e.g.

```
if (month==1 || month==3 || month==5 ...)
```

(and then either 2 `elseif` bits, or 1 `elseif` and one `else` in the `if ...` structure).

To test your understanding ... for the matrix:

$$A = \begin{pmatrix} 4 & 66 & 13 & 42 & 36 & 14 \\ 33 & 4 & 0 & 28 & 11 & 22 \\ 18 & 26 & 7 & 1 & 5 & 19 \\ 12 & 9 & 23 & 30 & 7 & 2 \\ 0 & 0 & 2 & 0 & 15 & 33 \\ 14 & 42 & 17 & 27 & 8 & 0 \end{pmatrix}$$

determine ... NOT using `find` (or similar), but rather via a nested loop, how many occurrences there are of values [5] [6]:

1. greater than `9`
2. greater than `9` but less than `20`

Make a new *script* **m-file** for this. You'll need to create a nested loop to test each and every location in the *array* in turn. Display (`disp`) the result at the end (after the (nested) loop has ended). You an either set fixed loop limits, e.g.

```
% define A
A = ...
% row loop start
for row=1:6
   % column loop start
   for column=1:6
      %%% CODE
   end
end
```

remembering that within the inner loop, you access that particular element of `A` by: `A(row,column)`.

Or you can be more clever/flexible and use the `size` function to determine the number of rows and columns on the basis that `A` might be any different matrix, e.g.

```
% define A
A = ...
% find size of A
[nrows,ncolumns] = size(A);
% row loop start
for row=1:nrows
   % column loop start
   for column=1:ncolumns
      %%% CODE
   end
end
```

which now becomes generic for any sized array `A`.

For e.g. the number of occurrences of values great than 9, you will need a counter, which you initially set as zero, before the first

[5] Hint: Before the next loop starts, you'll need to define a parameter to keep count of the number of values you find that meet the criteria, and set it to zero. Then in the (nested) loop, increment the counter variable by `1` each time you find a value matching the criteria.

[6] Also hint: At the start of the script (after your initial descriptive comments!), define `A`.

loop starts ... and then you increment ... i.e. add 1 to its value and re-assign the new total back to itself ... when the condition (value > 9) is met:

```
if A(row,column) > 9
   count = count + 1;
end
```

(This conditional test of whether the value of the current array element is greater than 9 or not, will go within the innermost loop.) Remember to set the initial value of `count` to zero (near the start of the *script* file (before the first (outer) loop starts)).

After the outer loop has ended, `disp`lay the answer (value of `count`).

Create a new *script* file and add the code above. Define a large-ish matrix `A` at the start of your program, filled with a variety of integers smaller and larger than 9. Run the program and check that it works (correctly counts all values in the matrix larger than 9). Try a different matrix and check that also works.

Note that you could always re-frame the *script* file as a *function*, and pass in the matrix `A` rather than re-defining it at the start of the program each time ...

NEXT: for the simple tic-tac-toe (3×3) grid, at each (*column,row*) location, you are going to draw a colored square.

Firstly, at the start of a new *script* **m-file**, add the lines:

```
% *************************
% YOUR COMMENTS ON WHAT THE PROGRAM DOES
% *************************
% create a new figure window
figure;
% create a set of invisible axes that will the window
fh = axes('Position',[0 0 1 1],'Visible','off');
% scale the axes (to go from zero to 3)
axis([0 3 0 3]);
% hold on!
hold on;
```

Here:

- The line starting `fh = ...` creates a plotting area with no axes visible, and filling the Figure window area (`[0 0 1 1]` in normalized units). The handle to this is returned (variable `fh`), just in case we ever need it later.
- Then, the axes are scaled for convenience – there are 3 rows and 3 columns in the grid we want to create, so a 'reasonable' choice is to set `axis([0 3 0 3])`, although we need not have.

- You know what `hold on` does, right ... ?

  You can then add the nested loop code framework[7]:

[7] To your **m-file**, after the `hold on` line.

```
% loop 1 start
for column=1:3
   % loop 2 start
   for row=1:3
      %%% CODE
   end
end
```

Note that for convenience and to relate things to a more familiar (x,y) coordinate system, we are now going to assume that `column=1` corresponds to $x=1$ in the plot, and `row=1` corresponds to $y=1$ in the plot. i.e. we are working from the bottom left hand corner, first across the columns, and then up the rows. (It does not matter what array location in terms of (*rows,columns*) you assume any (x,y) location corresponds to, as long as you remember what correspondence between (column,rows) in the matrix and (x,y) in the plot, you are assuming.)

To draw a square, the easiest function to use is `patch` (see earlier). For the coordinate parameters to be passed to `patch`, if your current location in the loop was `column=1`, `row=1` – assuming the notation and orientation where we start counting from the bottom left-hand corner – the coordinates for the bottom left hand square are:

$(0,0),(1,0),(1,1),(0,1)$

and for which `patch` will then take input:

```
patch([0 1 1 0], [0 0 1 1],'black');
```

(remembering that patch takes a vector of all the x-coordinates as a 1st parameter, and then a vector of all the y-coordinates as the second parameter).

If you do no more than this (and use the `patch` line exactly as written above in your code and within the inner loop), you end up looping through the (3×3) grid, but only even (re-)plotting the same square in the bottom left-hand corner ... this is 'OK' – exactly what you have told the computer to do so far.

The mental leap is to generalize the problem and to notice that if your column (x) and row (y) values correspond to the loop variables `column` and `row`, respectively, you could write:

```
patch([column-1 column column column-1], ...
   [row-1 row-1 row row],'black');
```

Take a while to think about this – work through in your head what would happen for `column=1` and `row=1` (this is the same as above).

Then ... what would happen for the next step in the loop – when `column` remains `1`, but `row=2`. Where is this square plotted? (Draw on a piece of paper or test in the code.) And then for `column=1` and `row=3`? Once we hit the end of the inner loop (`row=3`), the value of `column` is incremented by `1` and the `row` count is reset to `1` – where then would the `patch` square for `column=2` and `row=1` be drawn?

Play this mental (or on paper) game until the end `column=3` and `row=3` or until you are happy how the code works and how each of the squares in the (3×3) grid are going to be drawn in turn.

Try this (adding this `patch` command to your code). You should end up with 9 black squares in a (3×3) grid ... a huge black square filling the figure window, as per Figure 4.6 ... :o)

Figure 4.6: 3x3 grid of black squares ...

You could make it a little more interesting by creating a color value derived from the values of the `column` and `row` counters, e.g.

```
color = (column + row);
```

(on the line immediately before `patch`) and then modify the patch command:

```
patch([column-1 column column column-1], ...
   [row-1 row-1 row row],color);
```

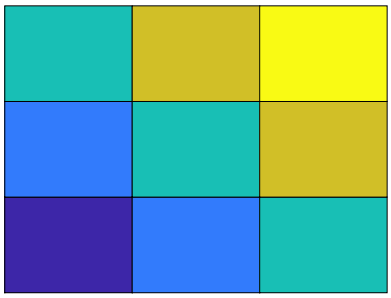

Figure 4.7: 3x3 grid of colored squares.

Re-running the script, you now get Figure 4.7.

You might notice that the colors are the same along diagonals, because you get the same value of `color` whether you are at location $(1,2)$ or $(2,1)$. We could make the location colors more distinct by modifying how we derive a value for `color`, e.g.

```
color = (column^2 + row);
```

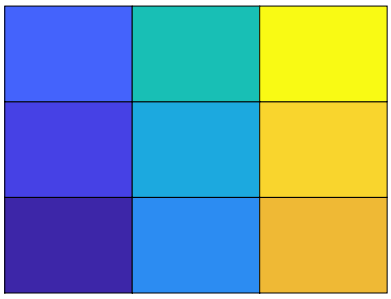

Figure 4.8: (yawn)

(Figure 4.8)

You could also create distinct colors by using the `rand` function (see `help` and Box). e.g.

```
color=[rand rand rand];
```

where we are now specifying random values (in the range 0-1) for each of the three red, green, blue color intensity components (see earlier). (No figure example shown.)

Maybe play about a little creating different patterns of colors.

`rand`

`rand` (with no passed parameters), returns a quasi random real number, in the range $0.0 - 1.0$.

This can be scaled, so e.g. `10.0*rand` returns a number in the range $0.0 - 10.0$.

`1.0+9.0*rand` returns a real number in the range $1.0 - 10.0$.

`round(0.5+9.999999*rand)` returns an integer in the range $1 - 10$.

(Remember, that having obtained a random integer starting from 1, you can use this to index an array and hence ultimately, access different images at random.)

As a final example, consider the chess board. A chess board consists of squares in a 8×8 grid. The squares alternate black and white. To define 8 squares (points) along the x-axis on the bottom row, you'd write something of the form:

```
for m=1:8
   % SOME CODE GOES HERE
end
```

(taking `m` as the counter along the *x-axis*).

Now, if you wanted to define 8 squares up each column (the *y*-axis), at each and every *x*-axis value, you'd need to loop through all the rows. So you need a loop in e.g. `n`, inside the loop for `m`:

```
for m=1:8
   for n=1:8
      % SOME CODE GOES HERE
   end
end
```

Follow this through to satisfy yourself that for each and every value of `m` from `1` to `8`, `n` loops from `1` to `8`, and hence visits every point in turn of a 8×8 (`n,m`) grid.

Actually, now we have got this far, it is good practice to consider how we'd define the black and white squares. We'll assume that black is represented by `'1'` (*true*) and white by `'0'` (*false*) and create a board (array) of all white squares to start with, i.e.

```
board = zeros(8);
```

(Refer to **help** or earlier for the syntax for help on the function `zeros`.[8]) This array will be defined at the start of your program – after any comment lines and before the first loop starts.

If we start with a black square (`'1'`) at the bottom left, we could define an *algorithm* for creating the grid as: odd column number squares are black, as long as row number is odd, otherwise white.[9]

To implement this in code – as we loop through both column (`m`) and row (`n`) on the board, we will test for the column number being odd and row number odd, OR, the column number being even and row number being even. If *true*, the square is defined as black. The only tricky bit is to determine whether the row or column number is even or odd. We do this by testing whether there is any remainder after dividing by 2, using the function `mod`. i.e. if the number is divisible by 2 with no remainder, the number is even; if the remainder is 1, the number is odd.

A complete (but lacking in comments!) code might look like:

```
board = zeros(8);
for m=1:8
    for n=1:8
        if ((mod(m,2)>0 && mod(n,2)>0) || ...
        (mod(m,2)==0 && mod(n,2)==0))
            board(n,m) = 1;
        end
    end
end
```

Given that mod 2 of an integer can only be `0` or `1`, we could write:

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.

[8] You could alternatively write this:

```
board = zeros(8,8);
```

mod

Not ... the opposite of `rocker` (which doesn't exist in **MATLAB** anyway) but short for *modulo*. Wikipedia helpfully tells us:

"In computing, the modulo operation finds the remainder after division of one number by another (sometimes called modulus)."

Or in **MATLAB**-speak:

```
b = mod(a,m)
```

"returns the remainder after division of `a` *by* `m`*, where* `a` *is the dividend and* `m` *is the divisor"*.

It turns out that as long as a is positive, you can use to test for whether an integer `a` is *even* or *odd* by:

```
b = mod(a,2)
```

When the returned value `b` is `0`, `a` is *even*, and when `b` is `1`, `a` is *odd*.

[9] Look up a picture of a chess board to convince yourself that this works.

```
board = zeros(8);
for m=1:8
    for n=1:8
        if ((mod(m,2)==1 && mod(n,2)==1) || ...
        (mod(m,2)==0 && mod(n,2)==0))
            board(n,m) = 1;
        end
    end
end
```

(don't type this in yet ...)
Easier to use and see,is to expand the `if ...` into its individual parts and add comments, e.g.

```
board = zeros(8);
for m=1:8
    for n=1:8
        % Here we are asking if both m and n are odd
        % => set value 1
        if ((mod(m,2)==1 && mod(n,2)==1))
            board(n,m) = 1;
        % Here we are asking if both m and n are even
        % => set value 1
        elseif (mod(m,2)==0 && mod(n,2)==0))
            board(n,m) = 1;
        % Otherwise, one must be odd and one even,
        % and we do not care which is which!
        % (set value to zero)
        else
            board(n,m) = 0;
        end
    end
end
```

(you are GO to type this in!)

Here, the final `else` bit is not strictly necessary as the `board` array has been initialized as being all zeros. (But it does not hurt to reassign a zero value if it helps you better read and understand the code.)

Spend a little time working through the code in your head and making sure you are happy how it works. It simply creates an array of zeros and then sets to a value of one, all the diagonally adjoining squares (as per Figure 4.9).

Create a new *script* **m-file** and add this code to it (plus comments!).

To crudely visualize the result (array contents), after the nested loop has ended (after the very last `end`), add a line using the `imagesc` function to plot the array contents (cf. Figure 4.9 and refer back to the syntax of how to visualize array contents using this *function*). Beautiful!

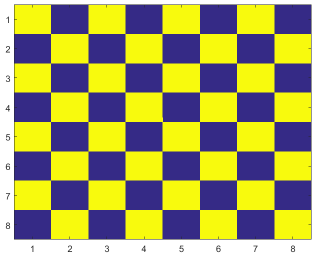

Figure 4.9: Chess board grid pattern.

You could also create a Figure – Copy and rename your working (8×8) grid code **m-file**. Comment out the `imagesc` line at the end. Then add to the top of your file, the code given to you from the start of the tic-tac-toe (3×3) grid section.

To draw the different color squares, you will need to use the `patch` *function* again. You can use the same algorithm as you used before ... replacing `column` and `m`, and `row` with `n`.

The only complication is that you want to draw a square with one color if the value of `board(n,m)` is zero, and another if `board(n,m)` is one, although you could also 'cheat' and simply not draw any square if `board(n,m)` is zero and that would become the white squares in the board (so then you only have to test for `board(n,m)==1` and draw a black square using `patch` if `true`. Whichever you choose – your `if ... end` structure will come still within the inner loop, but after the `if ... end` structure that sets the value of `board(n,m)`.

4.2 *Algorithms and problem-solving*

This (algorithms[10] and problem-solving) is not something that can really be 'taught' *per se*, but rather practised and aided by a logical state of mind. We'll go through a series of step-by-step examples. Hopefully this will also illustrate some general coding approaches.

[10] *algorithm* – "a process or set of rules to be followed in calculations or other problem-solving operations"

4.2.1 *Example #1: max(!)*

So yes, this is a built-in *function* `max` in **MATLAB**, but suspend disbelief for a moment ... and pretend that there is not one. What if we wanted to create one, i.e. a *function* that is passed a vector of numbers, and returns the maximum value on that vector? [11]

[11] **Example codes provided**

So already, from the definition of the problem, you know how to create a new **m-file** a define a *function* – one that takes as input a variable (a vector), and returns the largest value in that vector.[12,13]

[12] Note that you do not need to specify that the input variable is a vector, just that there is a variable input.

[13] Here, my personal naming convention has:
`s_out` – the output variable, and `s` for scalar
`v_in` – an input variable, with `v` designating vector ...
(This is perhaps, overkill, but leads little room for any confusion later.)

```
function [s_out] = maxx(v_in)
% maxx
%
% Takes a (single) vector as input;
% returns the maximum value.
%%% CODE
end
```

You have hence created a (empty) shell for the program (*function*). You could try calling it/running it at the command line, just to check there is no problem so far, although it is clearly not going to do anything. To call your *function* at the command line – remember that you have defined the *function* to take 1 input – you must therefore give it an input ... First – make a test input in the **MATLAB** workspace (not in the *function*), e.g.

```
» A = [1:100];
```

which defines a vector of integers between 1 and 100. You can now call the *function* at the command line, passing in this test vector to `maxx`:

```
» maxx(A)
```

Nothing gets returned yet, but equally, you should not be presented with pages of red **MATLAB** error text.

You could extend your basic testing by returning just the first element of the input vector, i.e., before `end`, add:

```
s_out = v_in(1);
```

and then re-try:

```
» maxx(A)
```

You are still not solving the problem yet, but in this step, you have now demonstrated that your *function* can take in a (*vector*) input, do something with it, and now set the output variable (and return a scalar)[14]. This is an important step in developing more complex programs.

OK, so ... we need to devise an algorithm to find the largest value in the vector `v_in`. The first piece of potentially useful information you can find, is the number of elements in the vector. You can obtain this via the function `length`[15].

So near the top of the *function* (but below the function definition line and the following comment lines), you could create a *variable*, set equal to the number of elements in the *vector* that you are going to have to process:

```
nmax = length(v_in);
```

What about the next part or structure in the program? You are going to need to search through all the elements of the vector if you are going to find the maximum value, so presumably a *loop* is required – one that loops through from the first to last element of the vector:

```
for n = 1:nmax,

end
```

(and which goes after the value of `nmax` has been defined).

The crux of the problem is recognising that you need to keep tabs on the running maximum value, or a local or temporary maximum value, that is your maximum value so far as you progress through repeated iterations of the *loop*. With this value, you are going to test whether each element of the array is larger than it – if true (the element in the vector being tested is larger than the current or largest-to-date maximum estimate) – you are going to replace the current maximum estimate with the vector element that you have just found is larger than your largest so far. We could call this *variable*, e.g. `temp_max` to indicate that is it temporary and not necessarily the largest value of the vector as a whole.

Within the loop, the test we make is therefore:

```
if (v_in(n) > temp_max),
   temp_max = v_in(n);
end
```

Make sure you are happy what this is doing – if the current loop value of the vector (`v_in(n)`) is greater than the largest value found so far as we work our way through the vector (`temp_max`), then we update the value of `temp_max` with it.

Almost there! Run the program/*function* and see what happens.

[14] Which in this example just happens to be the first element of the vector

[15] It turns out that `length` does not care about the orientation of a vector, and:

```
A=[1:10];
length(A)
```

gives the same answer (`10`) as:

```
A=[1:10]';
length(A)
```

MATLAB is unhappy about the line where the value of `temp_max` is being tested against `v_in(n)`. It may be obvious to you when (in terms of loop iterations) this is occurring. If not, why not, just after:

```
for n = 1:nmax,
```

`disp`(lay) the value of `n`. OR, add a breakpoint on the problem line, so that **MATLAB** will pause just before the line that gave the error, is executed.

Either way, you should have found that the value of `n` is `1`, i.e. the error is occurring on the very first iteration of the loop. Why? As per the error – **MATLAB** does not know what `temp_max` is. This occurs because you have not yet assigned it a value when the *loop* starts.

So our problem is one of initialization – we need to give `temp_max` an initial value, so that when the first iteration of the loop occurs, and the first element in the vector is accessed, there is something to compare it with.

There are 2 (probably 99999999) ways to go about this:

1. Seed the value of `temp_max` with a value so improbably small, that you are betting that any conceivable array of numbers will have a number greater than this.[16] For example, before the loop starts, you might write:

   ```
   temp_max = -999999999;
   ```

   And then go through testing all `nmax` elements in the vector.

2. Better, would be to initialize your temporary maximum variable with the fist element in the vector, i.e..

   ```
   nmax = v_in(1);
   ```

   You might also recognise that now, you need not test this against the first element, and the loop could start at `2` rather than `1`, e.g.:

   ```
   for n = 2:nmax,
   ```

Finally, remember to set the output variable equal to the maximum value that you find.[17]

There are 2 further testing or debugging (if you have issues) steps:

1. Firstly, simply go through in your mind, what you think happens on each iteration. Writing down how the values, e.g. of the temporary variable change on each iteration, is a good idea. Obviously you can do this prior to writing the code, to give you an idea of how it will work (or not).

2. Create a series of test *arrays* (*vectors*), or varying length, ordered, or random numbers, integers and/or reals, whatever youn like ... and see if your *function* works each and every time.

[16] There are obvious dangers here, should a vector of all insanely low values be given as an input. You could for instance determine whether any of the values were higher than the seed value, and if not, report or return an error message. So in this case, even if the function did not work, you would be told why. A bit like **MATLAB** functions in general, no?

[17] After the loop ends.

When you are passably happy with that – write a (new) *function* that finds the *minimum* element in a vector input.m[18] Call this function `minx`. **You will need this later on ...**

[OPTIONAL] – To deal with duplicate maximum values occurring – create a new (empty **m-file**) *function* that returns a second variable from the *function* – equal to the number of elements that are equal to the maximum.[19] (i.e. if the maximum value appears 5 times in the input vector ... you additionally return the number '5'.)

The structure of the (your new) *function* will now look like:

```
function [s_out1 s_out2] = maxxes(v_in)
% maxxes
%
% Takes a (single) vector as input,
% returns the maximum value
% PLUS the number of elements equal to that value.
%%% CODE
end
```

which passes back two variables (rather than the single one you had before).

You might try exploring what you are doing to 'do' with 2 outputs ... e.g.. if you (temporarily) added the following lines to your *function*:

```
s_out1=v_in(1);
s_out2=v_in(2);
```

then at the command line, you could test the *function*:

```
» maxxes([1:100])
```

here passing in a *vector* of integers from 1 to 100, and you should see:

```
ans =
    1
```

which ... is not right(!) and is only 1 of the outputs!!! So you need to tell **MATLAB** to explicitly assign the result of your *function* to 2 variables, e.g.

```
» [a b] = maxxes([1:100]);
```

Now you get the results you were expecting (`1`, `100`) returned and assigned to the variables `a` and `b`, respectively. Note that as you are returning result of your *function*, you can add the `;` to the end of the line to suppress unnecessary output at the command line.

Back to the construction of the algorithm and its coding ...

The code to generate the first variable (`s_out1`) – the maximum value of the elements in the input vector – is as before. You only

[18] Easiest is to copy the **m-file**, rename it `minx.m`, and edit the *function* name on the first line of the file. Then ... edit the code to find the minimum rather than maximum value.

[19] If we were cheating, and we are not ... then one could use the built-in **MATLAB** functions as so:

```
length(find(A==max(A)));
```

Which says ... find the elements in A, equal to `max(A)` and determine the `length` (number of elements) of this resulting vector (`find(A==max(A))`) ...

them need to work out the code to determine the value of `s_out2`. The key to this is recognising that previously, when the value of `temp_max` was equal to `v_in(n)`, you did nothing, as you were only interested in `v_in(n) > temp_max` and hence updating (replacing) the current value of `temp_max`. What you need is an `elseif` statement, and test whether you have found a second value in the vector equal to `temp_max`. If so, you need to somehow record this.

A partial solution (i.e. a first next step in the development of the code) might look like:

```
% Loop
for n = 2:nmax,
   if (v_in(n) > temp_max),
      temp_max = v_in(n);
      temp_n = 1;
   elseif (v_in(n) == temp_max),
      temp_n = 2;
   end
end
```

where `temp_n` is where we keep a track of the number of maximum elements – setting this to a value of `1`, when we first find a new largest value in the vector and then a value of `2` if we find a second occurrence of that value.

This does not quite work. Why? Try some test *vectors*, e.g.

```
A = [1 5 7 3 8 2 4];
B = [1 5 7 3 8 2 4 3 5 8];
C = [1 5 7 3 8 2 4 3 5 8 7 7 8];
```

In particular – is the answer to `C` ... right?

So far, we have accounted for a duplicate (maximum) value, but the solution (and code algorithm) is not *general*, i.e. we do not handle the general case of there being n duplicate maximum values in the vector. A more general solution looks like:

```
% Initialize max value
nmax = v_in(1);
% Initialize duplicate counter
temp_n = 1;
% Loop
for n = 2:nmax,
   if (v_in(n) > temp_max),
      temp_max = v_in(n);
      % Reset the duplicate counter
      % as a new max has been found
      temp_n = 1;
   elseif (v_in(n) == temp_max),
      temp_n = temp_n+1;
   end
end
```

Here we have to seed/initialize the value of `temp_n` because we start by setting `temp_max` as equal to `v_in(1)`, i.e. we already have 1 instance of the value of `v_in(1)` being the maximum, by the time the loop starts.

Missing from this code fragment is only the *function* definition and how the 2 *function* return values are set. See if you can complete the working *function* for returning the maximum value in a *vector*, and how many times that maximum value occurs.

Again – a key to programming and developing algorithms, is to follow the behaviour of the code in your head (as well as adding break points and testing with a variety of inputs, including extreme assumptions). For instance – assume that the 2nd element in the vector was equal to the first, and follow the code around – check that the value of `temp_n` is incremented appropriately and the output, if the vector was only 2 long, or there were no larger values in the remainder of the vector, is right. What if the 1st 3 elements were all equal? How does that pan out in terms of behaviour and output?

In general – if the code works for a selection of extreme assumptions, it will generally work. Use a combination of your head (and paper and pencil) and testing (and debugging if necessary).

4.2.2 *Example #2: sort(!!)*

(Yes, **MATLAB** also has functions for sorting values in an array ...)

In this second set of Examples – imagine that you have a vector of numbers, and you wish to sort them into ascending order. The *function* would take a vector as input, and return a vector of the same length as output, comprising all the values of the input vector, but now sorted in order.

How to go about this?

Well – first create the function framework (in a new **m-file**), as before:

```
function [v_out] = sortx(v_in)
% sortx
%
% Takes a (single) vector as input,
% returns a vector of the same length, with
% all the values sorted in ascending order.
%%% CODE
end
```

What do we need to do within the *function*? Well, we should start with a simple test of the *function* structure. For instance, we could start by finding the minimum value in the array, and placing it at the start of a new array – the one that will form the output. In this test,

in place of `%%% CODE` in the empty *function* outlined above, would go the following:

```
% Initialize the output vector (as empty)
v_out = [];
% Find minimum value
x = minx(v_in);
% Update output vector
v_out = [v_out x];
```

remembering that one way to build up a *vector* for output, is to append to (concatenate) a value to an existing vector – the `v_out = [v_out x]` bit. Also note that the vector must start defined as something ... here, we have created an empty vector `([])`.[20]

(**Make sure you created the `minx` *function*!!!**)

Try this out at the command line to check that it does indeed return the minimum value of the vector input, e.g.:

```
» sortx([1 2 3 4 5 6 7 8 9])
```

Of course, you could create a more complicated and challenging *vector* to test it with and rather than pass the values directly, e.g. if your *vector* of numbers to sort was assigned to *variable* `testvec`, then you would type:

```
» sortx(testvec)
```

So far, your *function* returns the lowest (1st) value, rather than sorting them all. You might spot that the 3rd through 6th lines of code, need to be repeated – i.e. you keep calling the find-the-minimum function, and adding this value to the output vector. But if this is all you did, the program would run forever and return an infinite number of repeats of the minimum value (or give you a vector the length of the input vector with identical (minimum) values in, depending on how you set up the loop (e.g. `for` or `while`)) – this is because once you have found the minimum value, and added it to the output array, the value still exists in the input array and will be found again (and again and again) if you create a loop around it.

The key ... at least, the key to creating the particular algorithm ... is to remove the element that you have just used from the input array. e.g. if we find a '`1`' and this is the lowest value, after it has been added to the output array, it needs to be deleted from the input array. This means that we need to know 'where' in the (input) array the minimum value was.

So ... your next task, leaving your *function* `sortx` to the side for a moment, is to modify your `minx` *function*, to return the position in the vector that the minimum value occurred at (in addition to the value of the minimum value itself).[21] i.e. your `minx` (or whatever you called it) function needs to look like:

[20] Also in this code fragment – you need to have created that minimum finding function ...

[21] HINT: When you find a new minimum value in the loop, the index (the position of that minimum value) is `n`. So you need to create a temporary variable that you update with the value of `n`, only when a new minimum value is found in the loop. You will see this variable with '`1`', consistent with assigning the first minimum value from index `1` in the vector.

```
function [s_out1 s_out2] = minx(v_in)
```

where `s_out1` is the minimum found value, as before, and `s_out2` is the index position of that number in the *vector*.

Get this working before moving on!

OK ... back to your *function* `sortx`:

Your use of `minx` now looks like:

```
[x p] = minx(v_in);
```

where in addition to passing back the minimum value and assigning it to the *variable* `x`, you also pass back the position of that value in the *vector*: `p`

You can now use this new information to delete this entry from the input array by means of:

```
v_in(p) = [];
```

The code within your `sortx` (not `minx`) *function*, should now look something like[22]:

```
function [v_out] = sortx(v_in)
% sortx
%
% Takes a (single) vector as input,
% returns a vector of the same length, with
% all the values sorted in ascending order.
%
% First:  determine number of elements in vector
nmax = length(v_in);
% Initialize the output vector (as empty)
v_out = [];
% Loop through all the elements in vector
for n = 1:nmax,
   % Find minimum value
   [x p] = minxx(v_in);
   % Update output vector
   v_out = [v_out x];
   % Remove used element
   v_in(p) = [];
end
% function end
end
```

Code this up ... test it ... play with it ... try and totally break it ... just **make sure that you can come to an understanding of how it works**.

For (easy) fun: create a (new) *function* that sorts in descending order.

OPTIONAL – For even more (but less easy) fun: create a (new) *function* that sorts in descending order ... but ... excludes duplicate values, i.e. no 2 values should be the same in the output vector.[23]

[22] Note the notation for catching the 2 returned values from the `minxx` function:

```
[x m] = minxx(v_in);
```

[23] HINT: modify the `maxx` function to return the index of all the elements having a value equal to the maximum value. This will be a vector, which will be appended to as the loop progresses, with each value being assigned the current value of `n` (the position of the maximum value in the vector).

4.2.3 *A gridded algorithm problem*

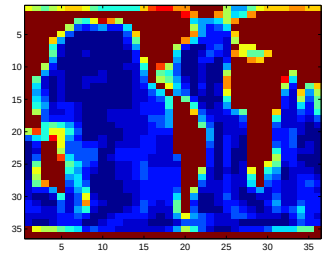

Figure 4.10: Ocean topography (blues through red) in the 'GENIE' Earth system model. Land is shown marked in brown.

We are going to base this next example around the (modern) topography of a simple Earth system model (**GENIE**).

Load in the file: model_grid.txt[24] 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[25], and values above 90, land[26] – it is possible to make out the shape of the continents visually in the pattern of numbers in the array (albeit they are rendered at low spatial resolution). The grid of numbers can also be visualized using the `image` function (see earlier). See if you can specify the scaling in such a way that you can render the ocean topography reasonably well, e.g. as per Figure 4.10.

What you are going to do is to draw this grid ... using the `patch` function. We'll simplify things and assume that each cell is 1 unit wide and 1 unit high – i.e. the grid goes from 0-36 units in both longitude (x-axis) and latitude (y-axis) directions. In fact – lets forget entirely about longitude and latitude for now.

Ultimately, the point of this exercise is to draw land as grey cells, and assign the ocean cells a color according to their depth. But lets start by drawing a grid of cells (of any color).

So how to start? Make yourself a new *script* (.m) file. I guess open a figure window, set `hold on`, and set the axes from 0-36 in both directions.

To begin with, simply draw a single cell (the first cell of what will become the grid). If should appear at the bottom left corner of the plotting area. If you find you have an odd shape appearing ... you have got a set of 4 y-coordinate values that is inconsistent (/out of sequence) with the y-coordinate values. Draw out the coordinates on a piece of paper and/or write down the 4 vertices of the rectangle, to help visualize.

You know that you will need to draw a row of 36, 1×1 squares using `patch`. So, make a loop ...

```
for i = 1:36,
   ...
end
```

(here: `i` for longitude). And then draw a series of squares, each with their right-hand edge corresponding to the value of `i` (so the left-hand edge is at `(i-1)`). For now, draw only a single row of cells, with the y/latitude-axis (which I will use index `j` for) from `0` (bottom edge) to `1` (upper edge).[27]

The key step here is to add in `i` into the list of x-coordinates, so that the x values progressively increase as the loop proceeds. (Leave the y-coordinate values alone for now.) Again – if you have odd

[24] From week #1.

[25] 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.

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

[27] To make the individual cells more apparent, you can specify a different edge color, and also make the edges thicker, e.g.

```
patch([0 0 1 1],[1 0 0
1],...
'k', ...
'EdgeColor','red', ...
'FaceColor','yellow', ..
'LineWidth',2);
```

(why it needs the 3rd, color option set, when the face and edge are set separately ... is one of life's little mysteries ...)

shapes appearing ... you have got an inconsistent sequence of coordinate values in the x and/or y vectors. Try substituting `i=1` into your list of 4 x-coordinate value and see whether you get the expected list of 4 (x,y) pairs. Then try substituting `i=1` in. (This can all be done on paper if you like.)

Now, you need to draw the other 35 rows of cells above this.

Think about this for a moment – you have a loop, drawing each of 36 square cells in a row .... in the first (bottom) row. Now you need to repeat the drawing of 36 rows ... a total of 36 times. This is a *nested loop*. Its form is:

```
for j = 1:36,
   for i = 1:36,
      ...
   end
end
```

meaning that whatever code goes in the very middle, is carried out 36×36 times (and in fact there are 36×36 cells in total to draw).

If your code to draw one line of cells was e.g.:

```
for j = 1:36,
   for i = 1:36,
      patch([i-1 i-1 i i],[1 0 0 1],'r');
   end
end
```

you now need to modify the y-axis values in the `patch` command, so that they reflect the increase in the value of `j` as you move up to a new row (and the `j` loop progresses towards a value of `36`).

See if you can get this working (just a 36×36 grid of cells ... colors of your choice ...).

Once you have this working: get a coffee.[28]

Actually, you won't need a coffee[29] – this is the hardest part done.

First, for the land cell designation. In the inner loop, rather than just draw a colored square regardless of anything, you need to decide whether to draw a grey (`[0.5 0.5 0.5]`) or e.g. red square, depending on whether the model grid at that particular (*i,j*) location is land or ocean. Land is designated by a value above 90. So you need to test the value of `model_grid`[30] – greater than 90 results in a grey square being drawn, and less than or equal to 90 (or just `else`), a e.g. red square. Try this, and if the grid comes out upside down, or back-to-front or something, you know how to transform the array you have read in.[31]

Second, to assign colors that depend on the depth. Rather than a e.g. red cell, when the cell is ocean (90 or less), create a RGB color that is a function of the depth value.[32] This is where the concepts of algorithms comes in – they need not be long, complicated codes, but

[28] You will need it.

[29] The margin note above was an alternative fact.

[30] Or whatever you called the array when you loaded the data in.

[31] You might note that while we tend to think about plotting of lon-lat as (i,j), in **MATLAB**, `i` corresponds to rows (lat) rather than columns (lon). So it is helpful to flip the rows and columns of the array around, so we can write (i,j) as (lon,lat) (i.e. (x,y). You might also find it is necessary to flip the array if it comes out up-side-down.

[32] In this particular model – depth goes from a value of 1 (deepest) to 16 (shallowest).

can be simple equations that achieve the desired result. For instance, given the nature of the RGB scale, and that we have a scale of values from 1 to 16, what immediately comes to mind is:

```
vcol = [1 1 1]/model_grid(i,j);
patch([i-1 i-1 i i],[1 0 0 1],vcol);
```

which has the effect of creating a grey-scale for the depth values from 1 (lightest) to 16 (darkest). The other way around would be:

```
vcol = 1.0 - [1 1 1]/model_grid(i,j);
patch([i-1 i-1 i i],[1 0 0 1],vcol);
```

This particular algorithm for converting depth to a unique color, will clash with the grey coloring of the continents (which were assigned a mid-grey) for a certain depth. So try and devise a color scale (in color!).

How about marking on the continental outline?

1. The first task is to draw the grid – as per above.[33]
2. Then, you want a second nested (i,j) loop, within which you will test for a boundary between land and sea, and draw a line to delineate this segment of coastline.

[33] You want to draw the complete grid first, because if you draw on the coastline lines as you go, you may find that you end up partially obscuring a coast line with the next filled cell.

There are a variety of ways to go about all this, some long with lots of duplicated code, and some cunning[34] and compact. We'll go for the ultra-crude approach, but leave it as an exercise for you to think about how it could be simplified and rationalized later on.

[34] So cunning in fact, that you could put a tail on it and call it a fox.

We'll take the case of the ocean being on the right hand side of a land (continental) cell (i.e. a East coast). We'll need to search through the entire grid and hence need a double/nested loop as before:

```
for j = 1:36,
   for i = 1:36,
      ...
   end
end
```

The plan will be[35]:

[35] Inevitably – you need to formulate a plan – your algorithm, first, whether simply in your head, or on paper.

1. Test for whether the cell is land (value > 90).
2. If the above is true, test for whether the cell immediately to the right, is ocean.
3. If the above is (also) true, then we have found a border between land and ocean and just need to draw the border.

You have done the testing of grid point (cell) values before, in coloring land one color and ocean (depth) another:

```
if (model_grid(i,j) > 90)
   ...
end
```

To then test the grid point to the right:

```
if (model_grid(i,j) > 90)
   if (model_grid(i+1,j) <= 90)
      ...
   end
end
```

Here – `model_grid(i+1,j)` is the cell to the immediately to the right (greater longitude) than `model_grid(i,j)`.

It is then just a matter of identifying the start and end of the `line` that you will draw.[36]

[36] Setting a thicker-than-default line width, e.g. `'LineWidth',2` will help the continental outline stand out.

The only one thing to note here, and if you have coded the loop as show above, you'll end up with an array out of bounds error reported by **MATLAB**. Think through what happens in the loop when the value of `i` reaches `36` – `i+1` is then `37`, yet the array is only 36×36. So in the case of finding the East coast segments of the continental outline, you need to:

1. Only loop from `i = 1:35`, so that the value of `i+1` is always a valid array index.
2. Because you still need to determine whether at the edge of the grid, there is an East coast line, carry out a specific test for the edge of the grid:

```
if (model_grid(36,j) > 90)
   if (model_grid(1,j) <= 90)
      ...

end
 end
```

Here – if the cell at the far right edge of the grid (`i=36`) is land, we test whether the cell at the far left of the grid (`i=1`) is ocean.[37] Note that this code fragment, because the value if `j` changes, goes within the outer, `1:36` j-loop (but not within the `1:35` i-loop).

[37] Remember that the grid wraps-around in longitude.

The complete code for this search ... except for the actual drawing of the edge line, is:

```
for j = 1:36,
   for i = 1:35,
   % Search i from 1 to 35
      if (model_grid(i,j) > 90)
         if (model_grid(i+1,j) <= 90)
            % DRAW EDGE
         end
```

```
        end
    end
    % Special case of i=36
    if (model_grid(36,j) > 90)
        if (model_grid(1,j) <= 90)
            % DRAW EDGE
        end
    end
end
```

It remains for you to create a similar code for finding (and drawing) the West coast segments. And then, the North and South coast segments. Remember in this latter search – the grid does not 'wrap-around' and j need only from `1:35` and `2:36` (with no 'special case').

In a final, optional, example ... of the bathymetry data – questions such as: 'How many land cells are there? What fraction of the Earths surface is land?', 'What (area) fraction of land is within 70 m of the current sealevel?', can be answered with 1, or at most, a few lines of code (and maybe a function call for the calculation of the area of a 1° grid cell). A more involved question might be: how many distinct land masses are there? Or: can we assign a label to them (assuming we want to in the first place).

Jumping straight into the full resolution 1 degree resolution dataset is probably not such a good idea, so instead, to start with, you are going to use the **GENIE** model grid/topography again. Further-more, you are only going to be concerned with the land-sea mask and not even worry about height above, or below, sea-level.

You are going to count up (and sequentially number) the different land masses[38]. Obviously, you could do this by eye for this particular example (but how about counting the unique land masses in the 1 degree topography dataset?). Think about how you are mentally 'doing' this – i.e. what processes are going through your brain (other than how long until the end of class) as you decide what makes any particular land mass distinct from another one. This may well inform how you go about coding and creating an algorithm to solve this.

A sensible start might be to loop through all the points in the grid. As you should have gathered – this can be done as a nested loop. To make it a littler cleverer: rather than setting in stone a specific count limit in the loops, which in this example would be 36 (for both longitude and latitude), you can extract the size of the array and hence the limits to the 2 dimensions by:

```
[n_lat n_lon] = size(model_grid);
```

[38] By 'different' – assume that distinct land masses (which here may be continents or just islands) are groups (or single) of land cells that share no common edges (excluding diagonal connections). The isolated block of cells representing Australia ia an obvious example.

Here: `size` returns the number of rows and columns of the array, corresponding to the number of latitude, and longitude bands, respectively. Your code (which should be placed in an `m-file`) will the start to look like:

```
topo = load('model_grid.txt','-ascii');
[n_lat n_lon] = size(topo);
lon=n_lon
   for lat=n_lat
   end
end
```

but with ... suitable comments added of course ... By all means add some suitable debug lines and test it (the loop behaviour).

You are going to need an array, the same size as the topography dataset, to store the number assigned to each land mass, i.e. each grid cell needs to be labelled with a land mass number, and something distinct from this if it is not land at all (i.e. ocean). You can create an array of zeros easily with the **MATLAB** `zeros` function (see Box). Then as you raster through the grid (via the nested loop), you can assign land points a value corresponding to the land mass number, and leave the ocean points as zeros.

To get your hand in – first add to the code above, the creation of the array of zeros (this is going to need to come after you have determined the size of the data array and hence the values of `n_lat` and `n_lon`, but before the loop starts). Then, within the loop, test for whether or not the grid point is land or ocean (see above for what the values in the **GENIE** model topography array mean), and if the point is land, set the value to `1`. Plot the results with `imagesc` and check that you get just `2` colors – one for ocean (`0`) and one for land (`1`). In fact, you could keep all this code and resulting array. Then for the array storing the land mass number, create a second array of zeros. (Remember to name the arrays something meaningful, not just `A`, `B`, `...`, and comment the code adequately.)

So how are you going to go about identifying new land masses and numbering them? You have to start somewhere, that somewhere will be designated by a `1` (the first land mass). How do you know that this is the first land point, and not the second? You could count up, for instance – each time you find a land point, you *increment* a counting variable by one, e.g.

```
n_runningtotal = n_runningtotal + 1
```

remembering that at the start of the code, you need to initialize the value of `n_runningtotal` to zero.

This is not quite what you want, for instance, if you run the following:

 str='do you like bananas?';

```
topo = load('model_grid.txt','-ascii');
[n_lat n_lon] = size(topo);
land = zeros(n_lat,n_lon);
land_id = zeros(n_lat,n_lon);
n_runningtotal = 0;
for lat=1:n_lat
   lon=1:n_lon
      if (topo(lat,lon) > 90)
         n_runningtotal = n_runningtotal + 1;
         land_id(lat,lon) = n_runningtotal;
      end
   end
end
```

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. [39] (It is quite pretty to plot though, as per Figure 4.11. Perhaps also try the 2 loops the other way around, with the `lon` loop first and outermost, and see what happens (/is different about it).)

As you might imagine, the crux of the algorithm is how to assign a new identifying land mass number to a land grid point only when it does not connect to a land point which already has a number – in this case, the same value for the identifying number needs to be used. In other words: if a newly found land point connects to a land point with the identifying value `5`, then the new point also needs to be labelled with a `5`. So ... and here is the critical bit ... we need to 'look around' each new grid point to see if there is an already labelled point immediately next to it. Pause and think about this. Maybe mentally, or on paper, work your way through the start of the grid, label the first land point you find, and work out what the mental steps are upon finding the next land point, to see if it needs to be assigned a new number, or not (and is instead connected to a point which already has a number). This mental/conceptual step is important and hopefully will lead you to a suitable and working *algorithm* that can be written down in code. In essence, all you are going to be doing is encoding (in code), using conditional tests and perhaps further loops, the mental steps that you are going through[40].

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.

[39] This is not a bad way of working in fact – get something of a likely correct form (e.g. nested loop in this case, setting up some arrays of zeros, creating a counter) but not quite getting the answer going first, then refine to get it doing what you actually want.

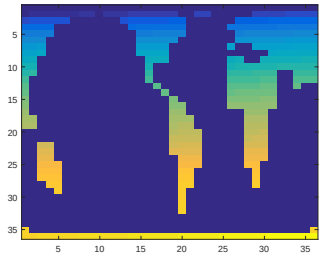

Figure 4.11: The 'GENIE' mode land grid, with land points assigned a sequential integer (working across and dow the grid – from West to East, and then North to South).

[40] Unless you are just thinking about `icecream`.

`icecream`
There is no `icecream` function in **MATLAB**. I checked. In fact, rather sadly, **MATLAB** tell me:

```
icecream not found.
```

So, *in practice* then, how are we going to decide if the counter is incremented and hence what value to assigned to a particular cell?

First, we need to test whether the current cell is ocean or land:

1. If ocean – do nothing, and leave corresponding value in the land mass array at zero.
2. Else (if land) – we need to work out what value to assign to the cell in the land mass array, by:
   (a) If an adjoining cell is land and has been assigned a value in the land mass array, then assign the same value to the current cell.
   (b) If all adjoining cells have a zero value, either because they are ocean, or because they have not been assigned a (non-zero) value yet (because the loop has not yet reached that far in the array), then increment the counter and assigned the cell this new number.

This simple decision tree is something that you could draw a flow-chart for if it helps. Also work through in your mind to see if it appears to 'work'.

The next step is coding the 'look around' (the current grid cell) bit. Actually, if you think about it, you need not look at the adjoining cells in all of the N, S, E, and W directions, because if we are looping through the grid such that we raster across the grid from left (W) to right (E), and then from the top (N) to bottom (S), cells to the E and S of the current grid point have not been reached yet and so must have a zero value. Hence you only need to interrogate the value of cells to the W and N of the current position (as defined by `(lat,lon)`). You can write the conditional test for the adjoining cells being zero (and hence ocean, as they must have already been visited and hence left with a zero value), by[41]:

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

[41] 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[42]. 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

[42] **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.)

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

```
if ( (lat==1) && (lon==1) )
  % on both Western and Northern edges (top LH grid
corner)
  CODE BLOCK #1
elseif (lat==1)
  % on Northern edge
  CODE BLOCK #2
elseif (lon==1)
  % on Western edge
  CODE BLOCK #3
else
  % cell lies neither on Western nor Northern edge
  CODE BLOCK #4
end
```

In 'CODE BLOCK #1', you will simply need to increment the land mass counter and assign the cell this value[43]. 'CODE BLOCK #4' will use the conditional code that you saw earlier:

[43] 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`).

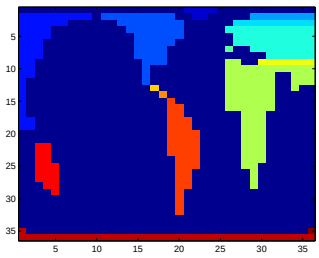

Figure 4.12: The 'GENIE' mode land grid, with land points assigned a unique identifier ... almost ... (!)

If you complete the code with the file loading and creation of the arrays of zeros, and then plot using `imagesc`, you should get Figure 4.12. Soooooo close[44]. Many of the continuous blocks of land have correctly been assigned a unique identifying number (the different regions of the same color in the figure). But something 'odd' happens in Eurasia, creating those stripes of color when it should be a solid block. It does not help to change the order of the loop (swapping the inner, `lon` loop for the outer, `lat` one) (Figure 4.13) and similar (but different – why?) artifacts arise (plus now one cell in Antarctica has a different color from the rest of the continent).

[44] Note that one could also question the decision to not count diagonal connections as representing continuous land. The result is that the single cell representing Spain and Portugal, is assigned a unique identifier. However, allowing diagonal connections would have the effect of joining North and South America.

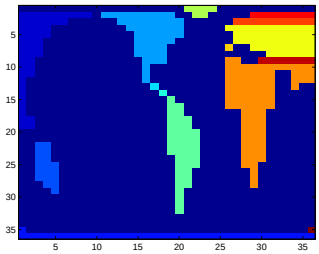

Figure 4.13: The 'GENIE' mode land grid, with land points assigned a unique identifier (color).

The way to debug this problem and write the code needed to adjust the algorithm is to again, work though in your head what happens when the loop is passing over the top of Eurasia. For instance, you can see that the first, mid-blue (value 4 in the `land_id` array) row is correct. But when the next row starts, because it starts at a lower longitude with ocean to the North, simply looking to the W and to the N does not reveal the existence of the row of 4s that start slightly later (in longitude).

As ever, there are a number of (equally correct) ways of correcting this. Here, we'll take the approach of post-processing the array, i.e. we'll leave the code that generates Figure 4.12 alone, but go back through the `land_id` array in a new nested loop, and fix the accidental partitioning of Eurasia into differently numbered strips. One possible solution is given below:

```
for lat=2:n_lat
  for lon=2:n_lon
    if ( (land_id(lat,lon)>0) && (land_id(lat-1,lon)>0)
...
         && (land_id(lat,lon) ~= land_id(lat-1,lon))
)
      old_id = land_id(lat,lon);
      new_id = land_id(lat-1,lon);
      land_id(find(land_id(:,:)==old_id)) = new_id;
    end
  end
end
```

In this, we skip the first row (Northern-most latitude) and first column (Western-most longitude) completely, because one might suspect that these grid points cannot be incorrectly labelled (why?), hence the `2:n_lat` and `2:n_lon` loop limits. The issue we are having and why the previous algorithm did not fully succeed, is that some of the land masses have been split into sperate strips, where adjacent cells sharing the same longitude, have different index values. i.e. we need to look for grid cells which have a different index value to the cell immediately to the North, as long as neither is ocean

(0). The way I have structured the if statement is to test for both lat and lat-1 cells not being 0, AND the two cells not being equal (i.e. having a different value). The result of applying this corrector code is shown in Figure 4.14.

Finally ... the longitudinal edge of the domain is also creating a problem, and land, which should be continuous across the longitudinal domain boundary is instead treated as separated (i.e. the Eastern edge of Eurasia on the LH edge of the plot is one color, but the rest of Eurasia (RH side) is another ... We can fix this by adding one further correction:

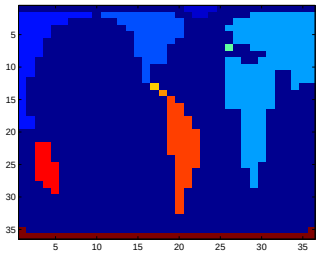

Figure 4.14: The 'GENIE' mode land grid, with land points (almost) assigned a unique identifier (color).

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

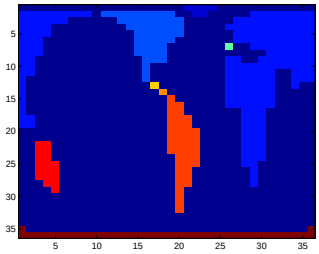

Figure 4.15: The 'GENIE' mode land grid, with land points assigned a unique identifier (color).

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

[46] 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[45]. 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. [46]

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.

```
178  str='do you like bananas?';
```

4.3 *Interpreting equations (0) – Basics*

4.4 Interpreting equations (1) – Population models

4.4.1 Exponential (and unrestricted) growth

Consider the simple mathematical population model[47]:

$$P_{(n+1)} = \lambda \cdot P_{(n)}$$

This defines the number of individuals in the population that there will be at some point in the near future, based on the number at the current time, where:

- $P_{(n)}$ ... is the size of the population at generation (or time) n.
- $P_{(n+1)}$ ... is the size of the population at generation $n + 1$.
- λ ... is the average number of offspring produced, per adult per generation, less mortality.

[47] 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).

Don't get put off by all the Ns and $_{subscripts}$ and things. All the equation says is that the population size (number of individuals = P) at the time of the next generation ($n + 1$) is equal to the population at the current generation (n) multiplied by some factor. This factor is given the Greek letter λ.[48] 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.

[48] We could equally write this in terms of time and if the units of λ were per year (yr^{-1}), rather than generation number n we would have time t (years since the start (of the model)).

So, we are simply asking; how many individuals will there be at the time of the next generation ($n + 1$)? The answer is; the same number as currently, minus the fraction of the population who snowboard off of a cliff or die of old age, $\alpha P_{(n)}$, plus the number of births in the population, which is also assumed proportional to the current number of individuals in the population, $\beta P_{(n)}$.

If there are P individuals in the current generation, the number at the next generation can be written:

$$p_{(n+1)} = p_{(n)} + \beta \cdot p_{(n)} - \alpha \cdot p_{(n)}$$

In code, this would look like:

```
P = P + beta*P - alpha*P;
```

Re-arranging, we get:

$$P_{(n+1)} = (1 + \beta - \alpha) \cdot P_{(n)}$$

The only even faintly subversive thing that has happened to the original equation, is that all these factors have been included in the value of $\lambda = (1 + \beta - \alpha)$.

Simple, eh? Mostly, that is about all there is to computer modelling. You know how much stuff (rabbits, snowboarders, cloud

water droplets, whatever) there is currently (or at a specific point in time), and you want to predict how much there will be in the future, which you take to be one unit of time (time-step) away. You estimate the change in quantity (rabbits, snowboarders, cloud water droplets) that occurs over the course of one generation, and add it to the current quantity.

This model predicts that as long as $\lambda > 1$, the population will increase exponentially, generation by generation, without end. Think of bacterial cells dividing in a petri dish. On each subsequent generation (or time step) there will be twice as many cells as there are currently (assuming that all the cells divide into two at the same rate and there is no mortality of cells). The value of 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 in a `for` loop, e.g.

```
for ...
   P = P + beta*P - alpha*P;
end
```

or use a `while ... end` construction and ensure the expression evaluates to *false* when a set number of cycles of the loop is reached (you'll need to create a counter for this), or the model might end when a certain degree of convergence (on a solution) has been achieved – i.e. when from time-step to time-step, the change gets smaller and smaller each time and at some point gets smaller than some pre-determined threshold.[49] 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.

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

You'll probably want to plot the results[50] and so you may want to save the data of population number vs. generation or iteration (i.e. 2 columns of data and a number of rows equal to the number of iterations through the loop plus one (why?)). The `save` function can be used for this.

[50] 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

4.4.2 *Restricted growth (and an equilibrium state)*

In a variant of this ... one might consider that most plant or animal (or bacterial or snowboarder) populations do not behave like this – instead they vary around some average level. This is because birth & death rates vary depending on the size of the population. For example:

- When the population is large, there may be little food to go round and the birth rate falls (or death rate increases).
- Or, when the population is very small, all individuals may have access to as much food as they can eat giving a high birth rate (or low death rate). For the bacteria in a petri dish, the population cannot go on expanding for ever – sooner or later the entire surface of the nutrient agar will be covered, leaving no free space for new cells to sit happily directly on the food. Later, the nutrients in the agar might start to become depleted. Toxic waste products might also start to build up, slowing down the rate of growth and cell doubling in the bacteria.

We can include a density-dependence by modifying the original equation, to give:

$$P_{(n+1)} = \frac{\lambda \cdot P_{(n)}}{(1+a \cdot P_n)^b}$$

There are two new parameters here:

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

Try starting with values of:

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

and run for e.g. 100 or 1000 generations (or however you are counting the loop in units of). Then systematically investigate the effect of changing the value of parameter b on the dynamics of the population, keeping the values of the parameters λ and a constant.[51] Increase the value of the parameter b and investigate how the dynamics change. Try values of b in the range 0.1 to 10. Try and find the approximate range of values of b that give the following types of dynamic of the population:

1. **Monotonic Damping** (smooth approach to a stable equilibrium).
2. **Damped Oscillations** (oscillates to start with then dampens down to an equilibrium).
3. **Stable Limit Cycles** (regular pattern of peaks and troughs with the population repeatedly returning to exactly the same size).
4. **Chaos** (population bombs about all over the place with no regular pattern).[52]

[51] 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.

[52] 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.

 str='do you like bananas?';

Don't spend too much time playing. I know how much fun you are having ;) The key take-home message is to recognise that the population value at each subsequent generation or iteration ($n + 1$) depends directly on the value at the previous one (n).

Here you are using a numerical model to explore how a system behaves, and how sensitive the behaviour is to a critical parameter (b in this example). This sort of exploratory investigation can help you identify critical parameter values that have a profound (and maybe unexpected) effect – for instance, if parameter b related to something that was impacted by climate change, you might be able to determine the point in the future when climate change might make a population unstable. You might identify a certain population level as genetically viable (anything below this being un-viable). You might then be in a position to make recommendations about conserving this species. And all from just playing around with a computer model!

4.5 *Interpreting equations (2) – Pure lovely maths*

HERE, we are going to code up a graphical representation of the Mandelbrot Set – Figure 4.16, Figure 4.17, and see Box. But we are going to do this nice and gently, via a simplified example.

The **Mandelbrot Set**, is the set of complex numbers c, for which:

$$\lim_{n \to \infty} \left| z_{(n)} \right| \leq 2$$

where

$$z_{(n+1)} = z_{(n)}^2 + c$$

and

$$z_{(0)} = 0$$

which ... shares all of the characteristics of gobbledygook, and I probably haven't even defined it mathematically correctly ...

A rendition of the solution is shown in Figure 4.16 and zoomed-in, in Figure 4.17.

4.5.1 *Sequence convergence (in 1D)*

CONSIDER the mathematical sequence:

$$z_{(n+1)} = z_{(n)}^2 + c$$

Here – each successive, $(n+1)$-th value of z, is equal to the n-th value of z squared, plus c. We would write this in code:

```
for n=1:n_max
   z = z^2 + c;
end
```

where the new value of z is set equal to the previous value squared, plus the value of c. For the code to work – missing so far here is the initial value of variable `z`, as well as what variable `c` is.

We'll start the value for z of zero, and the code[53] would look like:

```
n_max=10;
z = 0;
for n=1:n_max
   z = z^2 + c;
end
```

Here, defining beforehand (at the start of the code) the number of iterations (`n_max`) that the loop will go through.

We are interested in whether, for a given value of c, the value of z grows ever larger and larger (without limit for ever), or whether it settles down and converges on some (finite) value.

You can hopefully see by inspection of the code (and/or equation), and trying out different values of c, that some values of c lead to the iteration converging, or remaining finite and small, while others lead to progressively larger values, apparently growing without limit. For some example (real number) values of c and the sequences of z they lead to, refer to Table 4.1.

We could sort through a range of values of c, and for each value of c, apply the equation:

$$z_{(n+1)} = z_{(n)}^2 + c$$

iteratively, carrying out a given maximum number of iterations. We could then determine for which values of c the sequence converges, and for which it does not, e.g. [54]

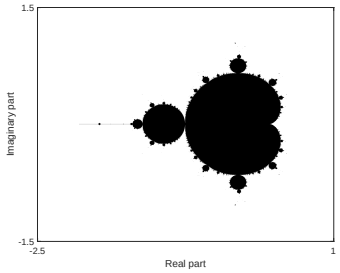

Figure 4.16: The Mandelbrot Set – points representing complex numbers that are members of the set, are shown in black. Complex numbers for which the sequence does not converge, are graphically represented by the white locations in the plotted domain.

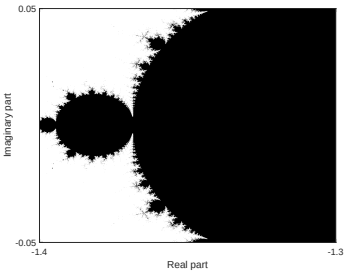

Figure 4.17: ×50 (-ish) zoom in on the Mandelbrot Set illustrating self-similarity and the fractal nature of the set boundary.

[53] Assuming just 10 iterations of the sequence.

[54] Instead of writing

```
z = z^2 + c;
```

faster is:

```
z = z*z + c;
```

value of c	sequence of values of z, as n increases (starting at n=0)
−3.0	0.0 → −3.0 → 6.0 → 33.0 → 1086.0 → ...
−2.0	0.0 → −2.0 → 2.0 → 2.0 → 2.0 → ...
−1.0	0.0 → −1.0 → 0.0 → −1.0 → 0.0 → ...
−0.5	0.0 → −0.5 → −0.25 → −0.4375 → −0.30859375 → ...
0.0	0.0 → 0.0 → 0.0 → 0.0 → 0.0 → ...
0.5	0.0 → 0.5 → 0.75 → 1.0625 → 1.62890625 → ...
1.0	0.0 → 1.0 → 2.0 → 5.0 → 16.0 → ...
2.0	0.0 → 2.0 → 6.0 → 38.0 → 1444.0 → ...
3.0	0.0 → 3.0 → 12.0 → 147.0 → 21612.0 → ...

Table 4.1: Examples of applying the equation iteratively (different starting values).

```
% clear workspace and close open figures
clear all;
close all;
% set (maximum) number of iterations to carry out
n_max=10;
% create sequence of numbers to test (vector)
v = [-3:0.1:3.0];
% fetch number of numbers in sequence (vector length)
n_v = length(v);
% loop through all the numbers in the sequence
for m=1:n_v
   % initialize (zero) value of z
   z = 0;
   % set value of c from vector
   c = v(m);
   % loop
   for n=1:n_max
      z = z^2 + c;
   end
   % assign result value depending on whether converged
   if (z > 2)
      v_conv(m) = 0;
   else
      v_conv(m) = 1;
   end
end
```

Here, after $n = 10$ iterations, the code tests whether the value of z has exceeded 2.0 [55] If the value of z has surpassed this threshold by the end of the 10 iterations, we are assuming that the sequence will never converge for this particular value of c. If not converging, the value of the vector `v_conv` (at the same index as the value of c was extracted from), is set to 0, otherwise, 1.

[55] Here, we are assuming rather simplistically, that 2.0 is a reasonable threshold for testing for convergence. It is somewhat arbitrary and a different criteria could equally have been used.

We could visualize the values of c for which it converges, via:

```
figure;
axis([-3 3 -1 1]);
scatter(v,zeros(1,n_v),50,v_conv,'filled');
xlabel('Value of c');
ylabel('n/a');
```

where I have created a dummy *y-axis*, with dummy (zero values). Each point is large and filled and colored according to the value contained in `v_conv`, which is either 1 (converging) or 0 (not converging after 10 iterations). The result is shown in Figure 4.18.

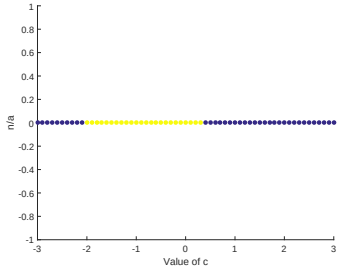

Figure 4.18: Solution space (blue points) for the simple sequence.

We can make the plot a little more interesting, by color-coding a measure of how quickly the sequence accelerates away to increasingly high values. For instance, we could color the points, if not converging, as some function of the highest value of z reached (when $n = 10$), e.g.

```
if (z > 2)
   v_conv(n) = 1/z;
else
   v_conv(n) = 1;
end
```

It turns out this is not very effective, as after 10 iterations, if not converging, generally very large values have been reached, and so in the color scheme, all non-converging values of `v_conv` are still close to zero.[56]

An alternative, is that for any given value of c, we identify how many iterations it takes to surpass the prescribed threshold (2.0) – the faster the sequence diverges, the fewer iterations of the loop will be needed to surpass the threshold. Now we cannot simply loop from 1 to 10 using a fixed `do` loop, because the value of 2.0 might be exceeded long before 10 iterations total has been reached.[57]

Instead, we could use `while`. A basic substitution of the current inner (`do`) loop would look like:

```
n = 0;
while (n <= n_max)
   z = z^2 + c;
   n = n+1;
end
```

Try this and satisfy yourself that it does exactly the same as before.

So now we add the additional criteria for terminating the `while` *loop* also test for the threshold being surpassed:

```
n = 0;
while ( (n <= n_max) && (z <= 2) )
   z = z^2 + c;
   n = n+1;
end
```

Now, the loop continues only if there are more allowed iterations (`n_max` has not been reached yet), and, the threshold has not yet been exceeded.

This code is faster than before, but your problem is pretty simple and you may not notice.[58]

The final step is to take the value of n that is reached when the `while` loop terminates, and use that to plot the color-scale.

[56] Nor does it help to simply set:

```
if (z > 2)
   v_conv(n) = z;
else
   v_conv(n) = 1;
end
```

(Try it and see!)

[57] Actually, we could use a fixed `do` loop, but it is much more efficient not to – if early on in the loop, the threshold has been surpassed, why keep iterating (and wasting CPU cycles)?

[58] If you would like to explore the efficiency of your program a little further:

1. At the very start of the code, add the line:
   `tic;`
   and at the end of the program, add:
   `toc;`
   Giving you a timing of the code execution.
2. Comment out all the lines of code for the graphics, so that you are left only with the calculations (and initialization).
3. Force the program to carry out a more challenging number of calculations, e.g.

```
v = [-3:0.000001:3.0];
n_max=100;
```

```
...
   % loop
   n = 0
   while ( (n <= n_max) && (z <= 2) )
      z = z^2 + c;
      n = n+1;
   end
   % assign value depending on whether converged
   if (n == n_max)
      v_conv(n) = 0;
   else
      v_conv(n) = 1.0-n/n_max;
   end
...
```

Here, the test for convergence of the sequence is out counting variable `n` having reached a value of `n_max` (i.e. for all the maximum allowed iterations of the loop, the value of z has remained less than or equal to 2.0).

Potting this now, looks like Figure 4.19.

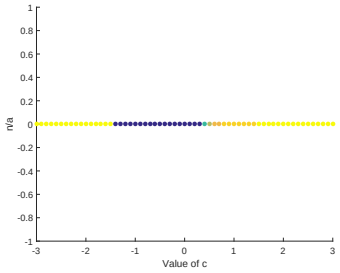

Figure 4.19: Solution space (blue points) for the simple sequence, with the rate of divergence forming the color scale of light blue (slowest) through yellow (fastest divergence).

4.5.2 *Sequence convergence (in 2D)*

Now to the Mandelbrot set ...

The idea is basically the same as before – we are going to generate a sequence, and find out whether it converges, or if not, how quickly it diverges and whizzes off towards infinity in value. The equation is very similar to before (see Box), with the next value equal to the current value squared, plus a constant, and we are varying the value of the constant.

The big complication is that c, is now not a simple *real* number (and one that we could simply plot along the *x-axis*), but a *complex* number (see Box).

It is helpful to think of the real and imaginary components of the number, as x and y values on an x-y plot[59] and treat them exactly as per you would vectors.

How to put this into code?

Well, the number c now has two parts – a *real* an *imaginary* part. Lets call them variables `x` (*real*) and `y` (*imaginary*).

The number z also has two parts. We could represent these by variables `a` and `b`. If we simply had the equation:

$$z_{n+1} = z_{(n)} + c$$

we could write (within a loop):

```
a = a + x;
b = b + y;
```

A *complex* number z, is a number of the form:
$z = a + bi$
where i is the square root of -1 (or $i : i^2 = -1$).
If we square z, we have:

$$\begin{aligned} z^2 &= (a+bi) \times (a+bi) \\ &= a^2 + a \times (bi) + (bi) \times a + (bi)^2 \\ &= a^2 + 2 \times a \times b \times i + (b)^2 \times (i)^2 \end{aligned}$$

and remembering what i^2 equates to:

$$z^2 = a^2 - b^2 + 2 \times a \times (bi)$$

[59] And in fact, this is exactly how we will be plotting things later.

For the equation

$z_{n+1} = z_{(n)}^2 + c$

we now have:

```
a = a^2 - b^2 + x;
b = 2*a*b + y;
```

(see Box). Except ... although we have taken the value of `a`, updated it, reassigned it back to the variable `a` ... when it comes to updating the value of `b` ... whoops(!) – we have already updated `a` (and we should not have as the original value is needed to calculate `b`). The simplest solution is to make the old and new values completely explicit:

```
a_old = a;
b_old = a;
a = a_old^2 - b_old^2 + x;
b = 2*a_old*b_old + y;
```

The equation above ($z_{n+1} = z_{(n)} + c$) in code, for `n_max` iterations (of n), looks like:

```
do n=1:n_max,
   a_old = a;
   b_old = a;
   a = a_old^2 - b_old^2 + x;
   b = 2*a_old*b_old + y;
end
```

Again, we could replace the `do` with a `while` as as before, apply a convergence criteria to terminate the loop (early):

```
while ( (n <= n_max) && ((a^2+b^2) <= 2^2) )
   a_old = a;
   b_old = a;
   a = a_old^2 - b_old^2 + x;
   b = 2*a_old*b_old + y;
end
```

or faster (but probably not important in this particular **MATLAB** program) would be:

```
while ( (n <= n_max) && ((a*a+b*b) <= 2*2) )
```

(because multiplication is faster for computers than raising a number to a power).

Mathematically, thats it. What remains is to create a set of values of c to test for convergence on, and because *complex* numbers can be represented in x-y space, we can create a 2D grid of real and imaginary component values, just as we did early for lon-lat values in plotting maps.

For example:

```
x = [-3:0.1:3.0];
y = [-3:0.1:3.0];
```

would create a range of *real* and a range of *imaginary* parts of the *complex* number $c = x + yi$.

However, as per for lon-lat, we want all combinations in a 2D grid, and so we use `meshgrid`:

```
[xx, yy] = meshgrid([-3:0.1:3.0],[-3:0.1:3.0]);
```

At this point – pause.

1. You have the loop framework code to test whether the maximum number of iterations has been reached, or whether the test of convergence has failed.
2. You have the code in the iteration loop, to square one complex number (`z`) and add a second (`c`) to it.
3. You have create a pair of matrices – one of values of a (`xx`) and one of b (`yy`) which together, map out a 2D space (/grid) to be searched.

Next, the full code will be provided to you (as an alternative to you trying to piece fragments together), but it is your job to make sure you understand it ...

```
% clear workspace & close open figure windows
clear all;
close all;
% create a parameter to contain the threshold value
thresh = 2*2;
% maximum number of iterations
n_max = 10;
% create initial grid ...
% from -1 to +3 in both dimensions, ...
% with a step resolution of 0.1
 [xx,yy] = meshgrid([-3:0.1:3.0],[-3:0.1:3.0]);
% determine total number of points to test
m_max = numel(xx);
% reshape x and y matrices into 2 columns of vectors
v(:,1) = reshape(xx,[m_max,1]);
v(:,2) = reshape(yy,[m_max,1]);
% create a 3rd vector column ...
% for storing a measure of convergence/divergence
v(:,3) = zeros(m_max,1);
% loop thought the x-y vector columns
for m=1:m_max
% set the value of complex number c
   x = v(m,1);
   y = v(m,2);
   % initialize z(n=0)
   a = 0.0;
   b = 0.0;
   % initialize the count
   n = 0;
   % iterate and check for convergence
   while ( (n <= n_max) && ((a*a + b*b) < thresh) ),
      % copy old value of z (n)
      a_tmp = a;
      b_tmp = b;
      % update z (n+1)
      a = a_tmp*a_tmp - b_tmp*b_tmp + x;
      b = 2*a_tmp*b_tmp + y;
      % update count
      n = n+1;
   end
   % set measure of convergence/divergence
   if (n <= n_max),
      v(m,3) = 1.0/n;
   else
      v(m,3) = 0.0;
   end
end
% take results vector, and ...
% reshape back into matrix form (for plotting)
zz = reshape(v(:,3),[length([-3:0.1:3.0]),length([-3:0.1:3.0])]);
```

In this code, you should note that I have avoided a double/nested loop for looping through the 2D space of the real (a) and imaginary (b) parts of the complex number c. Instead, I have simplified this to a single loop, of all elements. The total number of elements in the grid can be obtained using the `numel` function[60].

Knowing the total number of elements in the `xx` and `yy` matrices, it is a simple matter to convert these into vector form:

```
v(:,1) = reshape(xx,[m_max,1]);
v(:,2) = reshape(yy,[m_max,1]);
```

and then to add a 3rd column, that will hold the results:

```
v(:,3) = zeros(m_max,1);
```

You can plot the resulting grid of convergence/divergence values using `imagesc`:

```
imagesc(zz);
```

as per Figure 4.20.

To obtain a higher resolution plot – simply increase the resolution of the x and y vectors used by the `meshgrid`[61]. Also – the maximum of iterations allowed, `n_max`.

To create the simple black/white plot (e.g. Figure 4.16), I created a color scale which as all white, apart from black at the very start of the scale (corresponding to the lowest values). To do this, near the start of the code (before any of the loops), or, after both loops have finished, add:

```
% create color scale
brotmap = [ 0 0 0;
   1+zeros(9,3)];
```

and which looks like:

```
» brotmap
brotmap =
   0 0 0
   1 1 1
   1 1 1
   1 1 1
   1 1 1
   1 1 1
   1 1 1
   1 1 1
   1 1 1
   1 1 1
```

[60] In the code, it does not matter whether you write:

```
m_max = numel(xx);
```

or

```
m_max = numel(yy);
```

as the 2 matrices are exactly the same size.

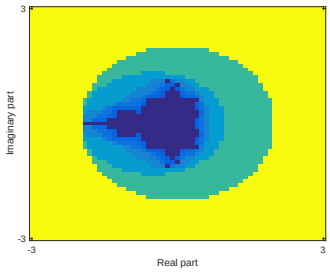

Figure 4.20: Simple, low resolution Mandelbrot set rendition.

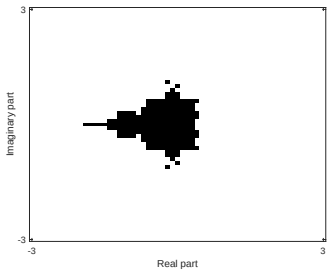

Figure 4.21: Simple, low resolution Mandelbrot set rendition (now highlighting points that are members of the solution set (black) vs. not (white).

[61] Note that you also have to make the same changes at the end when you reshape the results to the matrix `zz`.

defining black on the first row, and white on the next 9 rows. This gives a color scale of ten rows, that corresponds to the maximum number of iterations in your code, and hence the maximum number of different values that `zz` can take.[62]

Then, before you call `imagesc`, add:

```
cmap = colormap(brotmap);
```

The result is shown in Figure 4.21.

[62] Other choices for number of rows would have been perfectly acceptable in this particular example.

A more flexible way to define the grid limits and the resolution, is instead of writing in directly the specifications passed to `meshgrid`, e.g.:

```
[xx, yy] = meshgrid([-3:0.1:3.0],[-3:0.1:3.0]);
```

is to first set the grid limits:

```
x_min = -2.5; x_max = 1.0; y_min = -1.5; y_max = 1.5;
```

define the resolution – here the number of divisions:

```
xy_res = 2000;
```

and then for `meshgrid`:

```
[xx, yy] = meshgrid ...
   ([x_min:(x_max-x_min)/xy_res:x_max], ...
   [y_min:(y_max-y_min)/xy_res:y_max]);
```

This particular line is 'messier' than before, but now it is much easier to change the grid limits, and/or the resolution, and when you convert the results vector to a matrix, it is now just:

```
zz = reshape(v(:,3),[xy_res+1,xy_res+1]);
```

Try playing about with Mandelbrot Set plots – changing the x- and y-limits (`x_min`,`x_max`,`y_min`,`y_max`) as well as the resolution (`xy_res`) of the plot.

Figures 4.22, 4.23, 4.24 give examples of different regions (zooms)..

These example plots also employ a slightly more complicated color scheme:

```
brotmap2 = [ 0 0 0;
   jet;
   flipud(jet)];
```

which defines, as before, black as the color corresponding to the lowest values – in this case the solution set (a sequence that converges). But then it adds the built-in **MATLAB** `jet` color scheme to the end of this. And then ... for good measure, it adds on another copy of

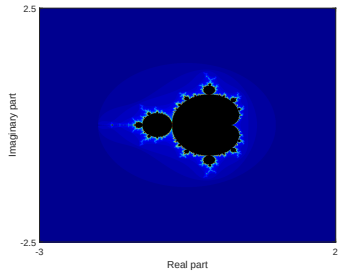

Figure 4.22: Initial Mandelbrot Set magnification.

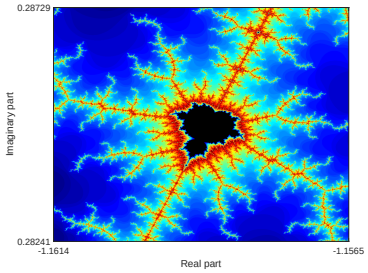

Figure 4.23: Example Mandelbrot Set zoom.

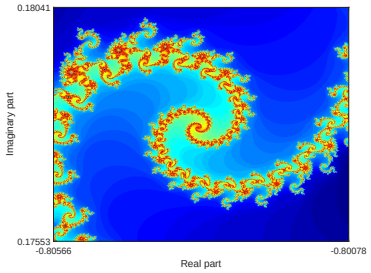

Figure 4.24: Example Mandelbrot Set zoom.

`jet`, but this time inverted[63] (colors occurring in the opposite sequence).

[63] The `flipud` accomplishes this inversion.

The real advantage of defining *x*-*y* limits it this way, is that you can re-formulate the code as a *function*, get the position of the mouse on the screen, and click to zoom by some fixed and predetermined amount, or to define a box to zoom to, and pass the new, updated x-y limits back to the *function* to re-calculate the sequence, and then re-plot the now zoom-ed in region of solution space.

In re-formulating your script as a function – you take as input, the *x*- and *y*-limits (four variables total), and return as output, an array of results values (`zz`). You would call this function from a *script* that does the actual plotting of the figure. The script would also set initial (default) *x*- and *y*-limit values and ... once the figure is drawn, take mouse input for a single click (to define the center of a zoomed in region) or two mouse clicks (to define the opposite corners of a zoomed in region) and then re-draw the plot.

Example code of a *script* (make_brot.m), and the corresponding *function* (fun_brot.m), are provided via the links.[64]

[64] Simply type:

```
» make_brot
```

to start, left mouse-button click to zoom to that point, and right-button mouse click to end (the plot window remains open however).

Zoom is controlled by the parameter `xy_mag` in make_brot.m.

5
Programming applications – games!

Games are great examples of many of the different facets of computer programming and **MATLAB** covered to date. They invariably contain algorithms and require problem-solving in the code, will contain multiple functions and sub-programs, loops, conditionals, graphics of some sort. They will invariably be complex, and hence put debugging skills to the test. They often even contain physics (and science)! They also provide an important motivation for developing the code – a specific and hopefully fun, end-product.

5.1 *Tic-tac-toe*

Tic-tac-toe[1] – Figure 5.1 – *'is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a* 3×3 *grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.'* It is as common a muck and inevitably, everypony knows how to play it, so we don't need to spend time defining the rules.

Here we'll devise a basic version requiring 2 (mostly) human players, but note the possibility of developing an AI computer player(!) (We'll also not make use of the **MATLAB** GUI, so keep the code as simple as possible, but note that as a further possibility for development.)

IMPORTANT: In the sections that follow – a number of code fragments are given to you. The idea is not simply to copy-paste the code fragments and go home ... The key is getting the structure of the program (and how the various sperate *functions* are created and utilized) right. If you find yourself having no idea 'where' to put a particular code fragment ... please ask! Note that you can always simply write your own code from scratch – there are many (infinite?) different ways of creating the program and writing the code to solve the different steps. You might even find that easier as it should be more obvious to you 'where' to put different lines.

A schematic of the complete (final) code structure is shown in Figure 5.2 as a guide.

To start – the following is a brain-dump on what we need to go about designing and writing the game[2,3]:

1. The game is almost wholly visual and so we need to think about the graphics at the outset. For instance:
   - We need a Figure window (no really)! Not having any axes/axes labels showing would be nice.
   - We need to draw a grid, consisting of two pairs of lines, at 90 degrees to each other.
   - We then need to add a cross or a zero to the graphic when a player clicks an empty square.
   - If there is a winner, we need to draw a line through the winning diagonal or row/column and the game finishes.
2. Associated with the adding of a cross or zero – we need to find a way to identify which box a player chooses, and then apply the cross/zero according to the selected box and player identity.
3. We need to keep a list (array) of the remaining empty boxes in the grid and only allow a box to be chosen if it is empty.
4. We need to test for a winning line of crosses or zeros.

[1] Also called 'Noughts and crosses'.

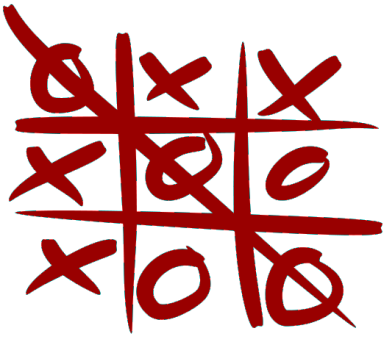

Figure 5.1: Tic-tac-toe. By Symodeo9 - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2064271.

[2] You can devise all sorts of strategies for creating the game, but you do need some sort of strategy before you start to write any code.

[3] Despite the messy, additional automatically-generated **MATLAB** code, some of these are actually easier in a GUI.

5. We also need some directions to be given to the players – who's turn it is, and who wins, or if there is a draw, that the game is over. To simply things, for now, these messages can be sent to the command line.

Some of these things you have seen and you will know (hopefully) how to go about it, such as drawing the grid, printing messages at the command line. Others are not so obvious and may prove tricky, so we'll tackle those first – to my mind, these are:

1. Selecting a location in the window.
2. Then filtering the chosen position to identify a specific box, and hence position the cross/zero neatly.
3. The mechanics of placing the cross or zero in the grid.

Before we go through these and progressively build up a working game, we'll start with a shell program *script* **m-file** – game.m – that we can test ideas and code in:

```
% ************************
% *** Tic-Tac-Toe game!  ***
% ************************

% close currently open windows
close all;
% clear variable space
clear all;

% create a new figure window
figure;
% create a set of invisible axes that will the window
fh = axes('Position',[0 0 1 1],'Visible','off');
% scale the axes
axis([0 3 0 3]);
% hold on!
hold on;
```

This is like you had before in drawing a grid graphic. You need not include `close all` ... but you may not wish to accumulate Figure windows for ever. Beasue this is a *script* m-file, not a *function*, the variables and their values remain in the **MATLAB** workspace even after the program is terminated or finishes. `clear all` simply ensure that a variable value from a previous test of the program, doesn't somehow interferer with the next test as you develop the code.[4,5]

The line starting `fh = ...` creates a plotting area with no axes visible, and filling the Figure window area (`[0 0 1 1]` in normalized units). The handle to this is returned (variable `fh`), just in case we ever need it later.

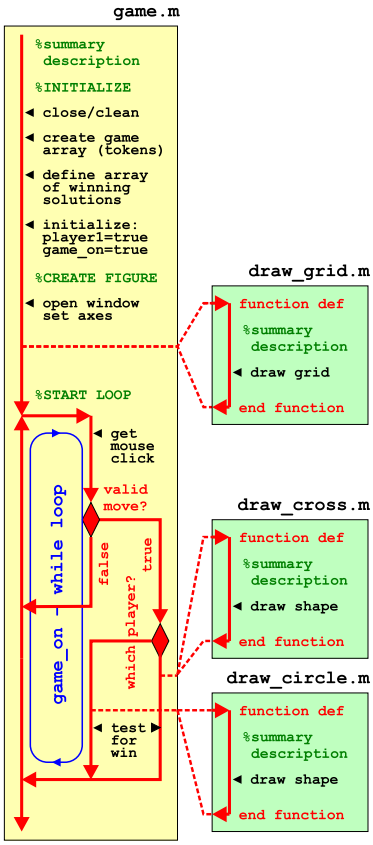

Figure 5.2: Schematic structure of the complete code.

[4] It shouldn't do, and there should be no variables used anywhere, that are not initialized to a specific value first.

[5] If you prefer to frame the program as a *function* ... with no inputs or outputs, then that is fine, but remember that you'll need to add *breakpoints* to interrogate any of the *variable* values as they all become *private*.

In scaling the axes – as there are 3 rows and 3 columns in the game area, it seemed 'reasonable' to set `axis([0 3 0 3])` , although we need not have.

Maybe before getting into any of the listed complexities, we could draw the game grid to give us some visual perspective on things. We could do this perfectly correctly, by adding the code to the main m-file/program file, but it is much neater to put sections of code that do specific things, and particularly if you might want to run these sections of code again, in a *subprogram*, which could be a *script* or a *function*. Here, even though there will be no inputs or outputs, we'll create a new **m-file** *function* (to be consistent with additional *functions* that we will be creating) just for the grid drawing – `draw_grid.m` – see Figure 5.2.

This is my *function* for drawing the grid (which will be called from the main program *script* file) and which is saved to a second new **m-file** draw_grid.m:

```
function [] = draw_grid()
%draw the game grid
grid_th = 2.0;
grid_col = [0 0 0];
line([1.0 1.0],[0.0 3.0],'LineWidth',grid_th,'Color',grid_col);
line([2.0 2.0],[0.0 3.0],'LineWidth',grid_th,'Color',grid_col);
line([0.0 3.0],[1.0 1.0],'LineWidth',grid_th,'Color',grid_col);
line([0.0 3.0],[2.0 2.0],'LineWidth',grid_th,'Color',grid_col);
end
```

(You should comment your version better!)

The 4 main lines of the code simply draw the 4 grid lines – 2 horizontal and 2 vertical. Because the line width and color appear 4 times – one in each `line` command line, I have set the value of a paid of parameters at the start – if I ever want to change line thickness and/or color, I need only make an edit in a single place (where the parameters are defined) rather than in each and ever `line` command line. (You could, for instance, experience with different line thicknesses and colors.)

To call the grid-drawing *function* `draw_grid.m` – in you main program (script m-file) – somewhere after `hold on` (refer to Figure 5.2), add the line[6]:

```
draw_grid();
```

which then calls the *function* to draw the grid.

Run it so far. It should look like Figure 5.3, depending on the line width and colors you choose in `draw_grid.m`.

[6] Remember you are not passing any parameters to this function, nor is it returning anything back to you

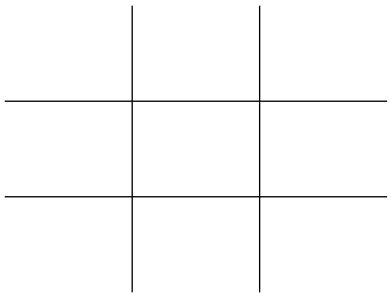

Figure 5.3: Tic-tac-toe game grid drawn.

5.1.1 *Mouse behavior*

OK – so a key part of the game is being able to select a particular grid square (cell), in order to place your (cross or circle) marker. One could do this e.g. at the command line by specifying a coordinate location, e.g. (1,1) for the bottom (or top) left cell, but this would be pretty tedious and would mean flipping back-and-forth between Command Window and Figure Window.

The **MATLAB** function `ginput` is provided to return the coordinate of the mouse pointer when one of the buttons is clicked. The coordinate returned is in the same units as your axes. Nothing is returned if you click outside the Figure Window.[7] `ginput` also returns which of the 2 (or 3) of the buttons was clicked. The `ginput` *function* also needs to be told how many mouse clicks to return – we need only one (per turn in the game).

To try this out (at the command line):

```
» [x,y,button] = ginput(1)
```

A figure window will open (unless you already have one open) and a set of cross-hairs appears indicating the current position of the mouse pointer. When you click a mouse button, it then sets the *variables* `x` and `y` equal to the (*x*,*y*) location of the mouse button click, and the *variable* `button` to the ID corresponding to which mouse button it was.

We'll use this in your game.m script shortly.

[7] Because we defined the game area as the entire area of the Figure Window, it should not be possible to click in the Figure Window but outside of the game area, so we do not have to deal with that possibility occurring.

`ginput` "*`ginput` raises crosshairs in the current axes to for you to identify points in the figure, positioning the cursor with the mouse ... `[x,y,button] = ginput(...)` returns the x-coordinates, the y-coordinates, and the `button`. `button` is a vector of integers indicating which mouse buttons you pressed (1 for left, 2 for middle, 3 for right).*"

So ... basically – you move the mouse and press a button, and **MATLAB** kindly tells you which button you pressed and where (within the axes) you pressed it,

5.1.2 *Drawing the 'objects'*

One strategy in programming, is to get *something* happening and working first, and worry about the details and quite what you really wanted, later. So lets draw *something* in response to the mouse click, and not yet worry what exactly we draw.

Again – creating subprograms and functions are a great way of reducing clutter in the main program, helpful in debugging, and all but essential if you need to apply the same (or almost the same) code more than once. The need to draw a number of crosses, and also a number of circles, fits the bill. So lets create a pair of functions for drawing first a cross, and then a circle. (See Figure 5.2.)

To begin with, the 2 functions will look almost identical, and differ only in name:

```
function [] = draw_cross(x,y)
%draw cross

end
```

for the cross (saved as `draw_cross.m`), with the other one:

```
function [] = draw_circle(x,y)
%draw circle

end
```

(saved as `draw_circle.m`).

Both function take a pair of parameters, `x` and `y` as inputs, which will be the (x, y) locations to draw the objects.

We should draw *something* (in the `draw_cross` *function*) to get things going. For now, try adding (in `draw_cross.m`):

```
dz = 0.25;
patch([x-dz x-dz x+dz x+dz],[y-dz y+dz y+dz y-dz],'b');
```

which is obviously not a cross (nor a circle) ... but it'll do for now. [8] Assuming the (x, y) location passed into the function is the centre of the object, `x-dz` and `x+dz` create x-coordinate vertices symmetrically either side (of `x`), and likewise for the y-coordinates. Experiment with a suitable value of `dz`.

You could test this simply at the command line ...

```
» figure;
» axis([0 3 0 3]);
```

and then call the function, passing whatever pairs of coordinates in the range $0 - 3$ that you like, e.g.

```
» draw_cross(0.5,2)
» draw_cross(1.5,1)
» draw_cross(2.333,1.333)
```

Create a similar shape for `draw_circle` ... picking a different color and/or a different shape. And then test this *function* (at the command line) also.

Lets recap at this point – you should be in good shape at this point – you have a main program (`game.m`), plus a function that draws the game grid (draw_grid.m) (and which is called from `game.m`), and ... you have a pair of *functions* to draw a shape centered on (x, y) (although so far they are not connected to anything else).

You also know how to find the (x, y) coordinates of a mouse button click (but have not added this to the main code yet.).

Now we come to the core of the game, which is that players can keep clicking the grid and choosing squares until someone winds (or you run out of un-claimed squares).

We are going to implement this with a `while ... end` structure (on the basis that something (code) keeps happening, until a

[8] You can also use `scatter` – plotting a single point, with a specific marker shape, to create the markers:
`scatter(x,y,5000,'o');`
would create a circle, and:
`scatter(x,y,5000,'z');`
a cross.
Here, the `5000` simply specifies a 'large' marker size.
To create a more prominent symbol and specify a particular color, add to the list of parameters passed to `scatter`, e.g.:
`'LineWidth',2,'MarkerEdgeColor','b'`

condition is met (end of game). So in your main program (game.m), add the following after you have drawn the grid:

```
game_on = true;
while game_on
   %%% CODE
end
```

You can test this by making **MATLAB** draw a marker symbol each and every time the mouse button is clicked. So in place of `% code`, add:

```
[x,y,button] = ginput(1);
draw_cross(x,y);
```

(and not fogetting to add your own illuminating comments!).

In this: having `game_on = true` without any line that could set the value of the *variable* to `false`, means that the *loop*, `while game_on`, loops ... forever. You'll have to CTRL-C to get out of this (or close the window), but if you click a number of times first, you start to get something that looks like if your luck was otherwise, this could be a 10 million dollar modern art piece (Figure 5.4).

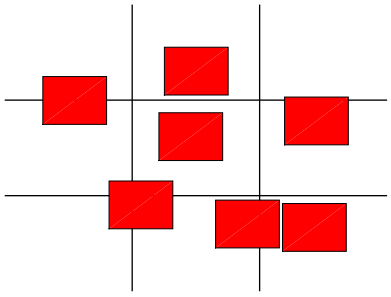

Figure 5.4: Tic-tac-toe game – object drawing test.

[**OPTIONAL**] As an experiment and test of your coding[9], try making the other shape appear if the other button is clicked – `ginput` returns a value of `1` to the variable `button` if it is the left mouse button, and `3` if it is the right. (See Figure 5.5.) i.e. in the code fragment above, rather than always calling `draw_cross` each and every time a mouse button is clicked – test for which mouse button it is and either draw a cross or a circle (or whatever shape is currently representing 'circle') depending on which it is.

Create a new **m-file** for this (by copying-and-renaming game.m), because you do not want this additional piece of code to appear in your final program.)

[9] If you need a hint – `if ...`.

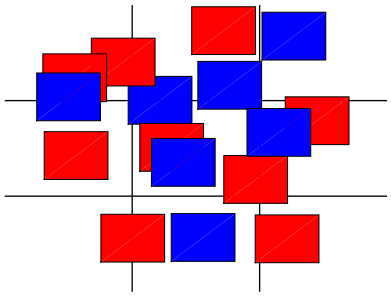

Figure 5.5: Tic-tac-toe game – object drawing + mouse button test.

5.1.3 *Identifying specific boxes*

There is still much to do ... but an obvious and significant next step is to place the objects in specific locations – i.e. centered in the box in which the mouse button occurred rather than simply at the exact same location of the mouse click. So we need to test the values of `x` and `y` (returned by `[x,y,button] = ginput(1)`), and identify a specific grid box (and its indices). [10]

Because you specified: `axis([0 3 0 3])` near the very start of your program, the grid is counted from the bottom left hand corner, from 0 to 3 in both x and y directions, so that the first row or column is in the range $0.0 - 1.0$, the second $1.0 - 2.0$, and the third $2.0 - 3.0$.

However, what we want is the index of the square, which will be 1, 2, or 3 (corresponding to the 1st, 2nd, or 3rd, column respectively).

[10] There is a simpler and sly-er way of doing this, which would be particularly useful if we have a really large grid and having 100s of `elseif`s is not practical.

The function `round`, returns the rounded up integer value of a real number. So `round(0.49)` returns `0`, while `round` returns `1`.

We could derive the value of `xi` simply, in a single line, by:

```
xi = round(x+0.5);
```

where the `+0.5` bit ensures values in the range $0.0 - 1.0$ returns `1` (and $1.0 - 2.0$, `2`).

We will call this index variable `xi` (for a variable containing the x index (or 'integer')) and could derive it from x by a piece of code such as:

```
if (x < 1)
   xi = 1;
elseif (x < 2)
   xi = 2;
else
   xi = 3;
end
```

(Work through this mentally, imagining a series of hypothetically clicked x-values between 0.0 and 3.0 and confirm that it will correctly give you the correct corresponding integer (1, 2, or 3).) [11]

You also need to write something similar for the y-direction.

Then – and add both these fragments of code (for determining the value of `xi`, and then for `yi`) to your main program immediately after the line:

```
[x,y,button] = ginput(1);
```

and before the line: `draw_cross(x,y);`

What you should not have is code within the loop, that determines the (x,y) location of the mouse click, and then convert the returned `x` and `y` values into a pair of (`xi`, `yi`) indices in the range $1-3$.[12]

[11] Or ... if you prefer:
Create a *function* with this code in, with one input (`x`) and one output (`xi`), and test passing different values of x in when you call it.

[12] Note that there is an easier way of obtaining the indices, using the **MATLAB** function `round`.

Try this out (run the program). You should have filled shapes appearing in a nice neat array, corresponding to each mouse button click, but ... not appearing in the center of the grid squares, but rather at the corners ... This is because we created the marker drawing functions assuming that the *(x,y)* values corresponding to the center of each shape were being passed. So we need to derive the center *(x,y)* coordinates of each grid square, and pass those.

Hopefully you can see that when we call `draw_cross` (or `draw_circle`), the center of the square which will be equal to the value of `xi` (and `yi`), minus 0.5, i.e.

```
draw_cross(xi-0.5,yi-0.5);
```

Modify your code like this and try it out.

Now ... suddenly ... the game seems to be coming together in terms of the graphics (Figure 5.6). (Obviously we are still missing a lot, including correct shapes.)

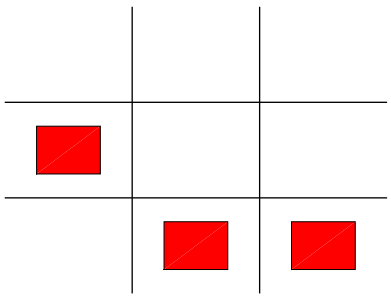

Figure 5.6: Tic-tac-toe game – object drawing now arranged in a grid.

Another re-cap.

What you code should look like so far is: after the `hold on` statement in the initial code framework (the very first piece of code given to start building the `game.m` program) is:

```
game_on = true;
while game_on
   [x,y,button] = ginput(1);
   ...
```

immediately followed by one set of code to identify the (integer) column (`xi`) number:

```
   if (x < 1)
      xi = 1;
   elseif (x < 2)
      xi = 2;
   else
      xi = 3;
   end
```

... and then one near identical set of lines of code to identify the (integer) column (`yi`) number following this:

```
   if (y < 1)
      ...
   end
```

And then:

```
   draw_cross(xi-0.5,yi-0.5);
```

Only after all this does the `end` of the `while` *loop* occur in your code.

```
   ...
end
```

You should be testing this code, which comprises the main (*script*) program `game.m`, plus 3 *function* **m-files**. It should run for ever, placing a 'cross' in the center of a square if whenever you click with the mouse button. Note that so far, you can add multiple squares in a row, and there is also nothing forcing you to alternative 'turns'. In fact, where are the 2nd player's markers?

5.1.4 Remembering turns (and arrays!)

A key to the game is that when a nought/cross has been placed somewhere, you cannot place anything more there. So we need to keep track of which cells have already been chosen. In fact, we need to keep track of what is 'in' each cell.

We will create a 3×3 *array* to store the information in, with each *(row, column)* pair of the **MATLAB** array, corresponding to an (`xi`, `yi`) pair (cell location in the game grid).

You have already seen how to create an $n \times m$ array of e.g. zeros. In your program – somewhere near the start, and certainly before the `while` *loop*, you can add[13]:

```
tokens = zeros(3);
```

(but adding a suitable comment line ...)

Another important thing you'll come across in programming is devising notations for representing states in a model or game or whatever. Pause and think about the possible states that each cell in the game grid can have.

1. Not yet chosen.
2. Assigned to a Cross.
3. Assigned to a Naught.

We could hence decide to assign values in the `tokens` array:

`0` == Not yet chosen.
`1` == Assigned to a Cross.
`2` == Assigned to a Naught.

(so the elements of the array can take a value of `0`, `1`, or `2`, as illustrated in Figure 5.7).

Now ... in the loop, as the `xi` and `yi` indices are derived in the code – use them to assign a value to the `tokens` array.[14] You can then test the value of the location that has just been chosen, and from this, decide whether the move is legal or not.

In your code, after having determined the values of `xi`, `yi`, and in place of the single line:

```
draw_cross(xi-0.5,yi-0.5);
```

add the following instead:

```
% test for square is empty (a zero value)
if (tokens(xi,yi) == 0)
   % if empty ...  draw cross ...
   draw_cross(xi-0.5,yi-0.5);
   % and then update array that is now has a cross
   tokens(xi,yi) = 1;
else
   % alert if square already taken
   disp('Illegal move!  Choose again.');
end
```

(here setting the value of `tokens` at that location to 1, because we are assuming still the 'cross' player in the variable and function naming notation).

Now when you run your program, when you click in a square, it always draws a cross, but only if there is not already one there.

[13] Or call the array variable something better ... there is no completely obvious and helpful variable name for what it will end up holding.

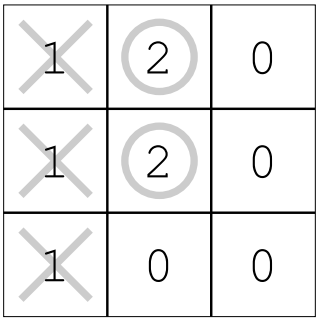

Figure 5.7: Tic-tac-toe game grid with numerical codes overlain.

[14] It is a personal preference whether to simply remember that **MATLAB** indexes arrays differently to reading a normal *(x,y)* location, or try and make the contents of the array, as viewed, look like the game grid.

5.1.5 *Putting it all together*

OK. Pause. Consider where you are at; what you have working ... and what remains to do.

Done:

- Drawn grid.
- Created functions to draw the 2 different game pieces.
- Recovered the *(x,y)* mouse click location, and converted that into the game grid *(x,y)* location.
- Checked to see whether a game cell is already occupied and not allowing the move if so (other placing a symbol).

To-do:

- Alternate the player turns.
- Draw 'correct' symbols(!)
- Test for the game finishing.

In terms of player turn – this is a simple binary state – either it is the turn of player #1, or it isn't (and hence the turn of player #2). So we could create a (*logical*) *variable* `player1` to keep track of whose turn it is.[15] If we initialize the game with player #1 starting first, near the top of the main program (before the `while` *loop*), we could set:

```
player1 = 1;
```

Now it is simple to alternate between the player turns, and after the current player has taken their turn, we can simply write:

```
player1 = ~player1;
```

which flips whose turn it is. This line will go within the `while` loop and after a player has taken a turn.:

```
if (tokens(xi,yi) == 0),
   draw_cross(xi-0.5,yi-0.5);
   tokens(xi,yi) = 1;
   player1 = ~player1;
else
   disp('Illegal move!  Choose again.');
end
```

(We do not change the player turn if it is an invalid cell (`else ...`) because the same player has to chose again.)

At this point, we are still not differentiating between the different players – we need to draw a different symbol depending on which player it is, and also set the corresponding element in the array to a different value (`1` for player 1, and `2` for player 2).

So we need to test for which player it is currently[16]. Now, in place of the 3 lines of code above, i.e.

[15] Equally, we could have defined a variable player, that took a value of `1` (for player 1's turn) or `2` (player 2). We would then need to change its value after a player had taken a turn, from 1 to 2, or 2 to 1. This turns out to be more awkward to implement than simply taking the *NOT* of a variable state.

[16] Note that we need only have one occurrence of the line `player1 = ~player1;`, although it would have still worked fine to have put this line at the end of the code in the `if` section, and also the `else` section.

```
draw_cross(xi-0.5,yi-0.5);
tokens(xi,yi) = 1;
player1 = ~player1;
```

we instead write:

```
if player1,
   draw_cross(xi-0.5,yi-0.5);
   tokens(xi,yi) = 1;
else
   draw_circle(xi-0.5,yi-0.5);
   tokens(xi,yi) = 2;
end
player1 = ~player1;
```

which tests for it being player #1's turn, and if so, draws their symbol and marks that cell (in `tokens`) as 'theirs'. Otherwise, draws player #2's symbol and marks the grid square as theirs.

Note that all this goes still within the `if` test of whether the move is legal or not, i.e.:

```
if (tokens(xi,yi) == 0),
   ...
end
```

If you test the code now, the output of the forced turn alternation should starts to look like Figure 5.8. The behaviour should be that you can only play a marker in an empty square, and the player turns alternate.

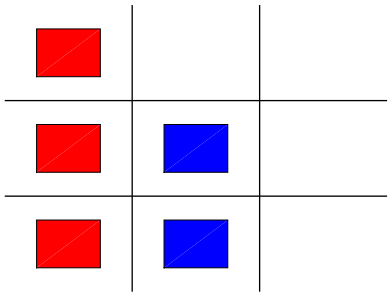

Figure 5.8: Tic-tac-toe game – object drawing now arranged in a grid and with forced alternation in player turn.

Finally, we need to think about the end-game. One way in which the game ends, is if there are no free cells left. We could:

1. Search through the 3×3 grid, testing each cell in turn as to whether it has a value of zero or not.
2. Or better – `find` – find the *vector* of indices of locations in which a value of zero occurs, and test whether this vector is empty[17]:

   ```
   remaining = find(tokens == 0);
   if isempty(remaining),
      break;
   end
   ```

   You could also add a message before `break`-ing out of the `while` loop.[18]

   This code would go just before the end of the `while` *loop* and either just before, or just after the line:

   ```
   player1 = ~player1;
   ```

[17] See Box. Note that if `find` finds nothing, it returns the empty vector `[]`.

[18] An alternative to the use of `break`, would be to set the value of `game_on` to false.

`isempty`
MATLAB says: '*Determine whether array is empty*', and:
`TF = isempty(A)` returns logical `1` (true) if `A` is an empty array and logical `0` (false) otherwise.

Almost almost almost there ... and perhaps the single most hardest part – detecting a 'win'.[19]

Probably, the easiest way, which would not be true for many other games, is to pre-define the every single winning pattern, and look through this list to see if any of these patters occurs. For instance, one winning pattern is shown in Figure 5.8 and would be represented matrix form by:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

We could define a series of 3×3 arrays to represent these. We'd end up with 8 different arrays and array names and that might be a bit of a mess to deal with. Better is to create a single 3D array, with the 3rd dimension having length 8 – one layer for each possible winning pattern.[20] If you did this (create a $3 \times 3 \times 8$ array ... we'll call it array variable `winning`), we could simply loop through the 3rd dimension of the array, from 1 to 8 (in terms of the layer number), and access each possible solution in turn. How to use this in practice? Well, it is not obvious.

A slightly different alternative is to use `find`, and for each player, obtain the list of '*linear indices*' (see below) of the grid cells containing their symbol.

For instance, for player 1 (red symbol) in Figure 5.8, we could create the matrix of the location of their marked squares and use it to explore strategies for determining whether it constitutes a 'win'. The matrix would look like this:

```
A = [1 0 0; 1 0 0; 1 0 0];
```

If you then do `find` for 1s (player 1's token value), you get:

```
» find(A==1)
ans =
   1
   2
   3
```

So for the winning 2D pattern in Figure 5.8, we could represent this more simply as `[1 2 3]`.

Be careful here – for 2D (or higher dimension) arrays, rather than the simple (1D) vectors you have been throwing at it previously – `find` returns the '*linear indices*' of the array locations where the condition is true. For a *linear index* – rather than giving a *(row, column)* index, **MATLAB** counts continuously, down the rows in the first column, then down the next column, etc etc, to give an index as shown in Figure 5.9.

If we define a winning pattern as its 3 linear indices:

[19] And don't panic ... because the code will be given to you at the end because it is not at all simple (although you might yet come up with a much better solution ...).

[20] If you like – make an analogy with the month temperature data, where you had 12 slices (the 3rd dimension) of a 2D *(lon,lat)* array of points.

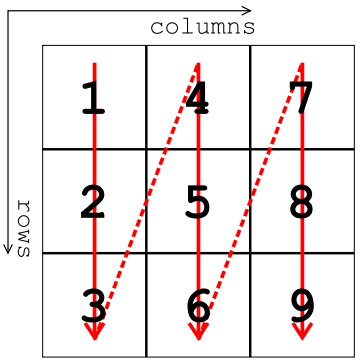

Figure 5.9: Linear indices of a 3×3 matrix.

```
winning = [1 2 3];
```

(for the first pattern), we can search the game grid at the end of the each player turn and look for whether this (winning) pattern occurs anywhere. We will use the **MATLAB** function `ismember` like this:

```
» ismember(winning,find(A==1))
ans =
   1 1 1
```

where the three `1`s indicate that each of the elements of `winning`, do indeed appear in the result of `find(A==1)`[21]. Only if three *trues* (`1`s) are returned, does the pattern completely match. To test for this condition, we can calculate the sum of the result of `ismember` and determine whether this is equal to 3. This then indicates to us that the winning pattern exists.

Your only job (as the code needed is given below) – is to define the 8×3 array `winning`, which will contain all the 8 different winning patterns (rows), in terms of a linear index (i.e. what `find` returns) (see margin clues).[22,23] For instance, the 3 different winning along-column patterns would be represented by:

```
winning = [1 2 3 ; 4 5 6; 7 8 9];
```

and you need to add (in the same array, `winning`), the 3 winning row patterns (in linear indices), plus the 2 diagonals, for a total of 8 different patterns of 3 indices.

Define this array (`winning`) somewhere before the *loop* starts.

The code for player 1, comes in the program (within the *loop*) just after you have set the value of the current cell in `tokens` to `1` (i.e. after player 1 has just taken their turn and you want to check whether they have just won):

```
pattern = find(tokens==1);
for n=1:8
   test_for_win = ismember(winning(n,:),pattern);
   if (sum(test_for_win) == 3),
      disp('Player 1 WINS!');
      game_on = false;
   end
end
```

(and similar code is needed for player #2 just after they have taken their turn). Note that if a winning combination is found, the game ends and `game_on` is set to `false`.

You could, of course, rather than write out this code twice, once for each player, create a *function*, passing in the player number, the current tokens array (`tokens`), and the winning patterns (`pattern`),

[21] The '`1`' because this is the notation for player 1.

[22] The way to go about it is to create a single winning pattern, and text the code and that it works, then define the remaining 7.

[23] Also – write down on paper, the linear indices of the 3×3 array – that will help, e.g.:

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

and perhaps returning a *logical* value representing whether a win had occurred.

It is also left up to you, to improve the shapes/symbols used to mark the squares of each player. The cross is relatively easy to draw. The circle is harder.[24]

Or use `scatter` and set the marker symbols and size appropriately.

[24] A polygon with a very large number of sides would do.

6
Graphical User Interfaces (GUI)

In this chapter we'll learn how **MATLAB** can create a simple *Graphical User Interface* (*GUI*), which you can used to interface to your program with (as an alternative to e.g. the command line). Scientifically ... this is less useful, but it is how all computer/device software/apps tend to 'work' these days and so is a good thing to learn about/how to do. (excepting devices (e.g. wifi routers) that use a web-browser for their interface, but then that is effectively a *GUI* within a *GUI* ...).

 str='do you like bananas?';

6.1 *MATLAB GUI basics*

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

1. It facilitates the design of the *GUI* window(s).
2. It creates a code framework for the associated program.

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

```
» guide
```

and a window as shown in Figure 6.1 should appear.

We'll only concern ourselves with the default option amongst the 'Blank GUI (default)'[2]. As for the tick-box 'Save new figure as:' – we'll leave this alone[3]. The 'Preview' window is blank at this point because you have selected a blank template (d'uh!) (and are not loading in a previously created *GUI*).

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

- Close/minimize/maximize buttons at the top right (and the window can be re-sized by grabbing the corner and dragging the mouse).
- A title at the top (in the title bar) with a cute (barf) **MATLAB** icon.
- 3 buttons at the bottom right – 'OK', 'Cancel', and 'Help'. Presumably they'll all do something (different) when clicked.
- Everything else is neatly enclosed in a pair of *tabs* (one labelled 'Create New GUI' and one 'Open Existing GUI' and you can switch between tabs by clicking on the required tab.
- In the 'Create Existing GUI' tab, there is:
  - A list (of GUIDE template names plus that annoying cute little icon again).
  - An area with a border labelled 'Preview' with a grey box labelled 'Blank' in the middle.
  - There is a *tick box* and next to it (grey-ed out by default), a box with a file path and name in and to the right of that, a button labelled 'Browse'.
- (In the 'Open Existing GUI' tab ... nothing much (yet) going on.)

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

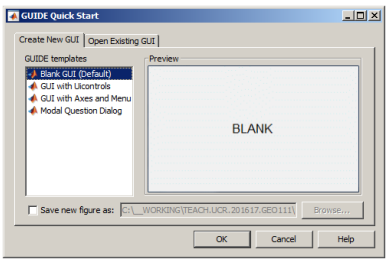

Figure 6.1: Starting GUI window of the **MATLAB GUIDE**, GUI design tool.

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

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

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

(If you have not already clicked OK in the GUIDE window – do it now.)

Rather than creating a blank m-file and/or some basic code first[5], **MATLAB** throws you straight into a window design tool as per Figure 6.2. There is a lot going on here, but start by noting there is the usual drop-down menu bar at the very top (under the title bar ('untitled.fig') of the window) and a row of icons underneath that (no re-appearance of the **MATLAB** icon thankfully). At the bottom of the window there is some information, mostly about location (of what?) – Current Point and Position. To the left of the window is a group of icons[6] plus a (depressed, by default) mouse pointer icon. Most of the window is made up of a pane (whose contents apparently is, or might be, larger than the area shown as indicated by the presence of scroll bars along the right and bottom edges). The pane itself is ruled with a grid pattern. (At least, that is what I see!)

Again – the great advantage of familiarity (of program *GUI* design) – you might guess (you'd be correct if you did) that the icons to the left allow you to select an object and place it in the pane, the grid serving to help you position the object. And this leads us to an important point – creating *GUI*-based programs is as much (or more) about design as it is about programming. The cleverest program (and most complex calculations) might simply be a total fail if the *GUI* is wholly unappealing or complete un-intuitive (or lacks a *GUI* entirely) and no-one wants to use the program (zero user-base). The grid is there for a reason and that is to guide you towards creating an ordered (and aligned), logical, and uncluttered arrangement of things (we'll come to what the 'things' are shortly) within the *GUI* window.

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

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

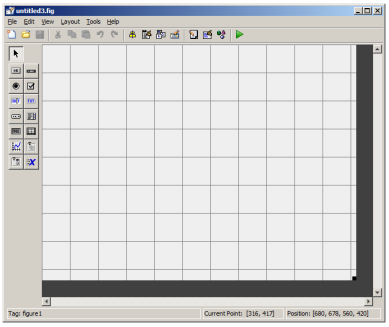

Figure 6.2: (Blank) GUI window editor GUI window.

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

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

[7] Which has a steep power relationship with the amount of code.

6.1.1 *Hello, World [Static Text (box)]*

This is as simple as it is going to get for a 'program' with a *GUI*. In the GUIDE window editor, which should be already open if you haven't fatally mucked about with it (or open up a new *GUI* now by typing `guide` (lowercase) at the command line again) – identify the Static Text icon (by hovering the mouse pointed over an icon, its function is revealed). Click (`left mouse button`) on it. The mouse pointer, when over the gridded design pane, should change to a cross-hairs mouse pointer icon.[8] Find a convenient place .... perhaps at the intersection of two grid lines, click the mouse down and drag out a box – this will be the size (and location) of the Static Text object. Release the mouse button to finish. If you don't like the size or location, you can move/re-size just like you would with a normal **Windows** (or **MacOS** etc.) window.

So far, the (static) text object has a rather unappealing default content of 'Static Text' in a small font. You can edit the properties of this object by double-clicking on it[9]. Whoa! That's a long list of ... actually, *properties* of the *object* (thats two new buzz-words in one sentence – *object* and *properties*). Each property (the column on the left) has a default value (the column on the right) assigned to it. Evidently, you can edit the properties using the design tool rather than in the code code, setting a parameter value.[10] For now, we'll just make two changes:

1. For the String property – click in the box to the right, delete 'Static Text' and write 'Hello, World'.
2. The text is pretty small ... so for the FontSize property, click in the box to the right, delete 8.0 and write ... well, try something larger.

Within reason, why not also play with some of the other properties if you like (at least, the ones that you can make a reasonably informed guess as to what they do). Maybe you end up with a design window looking like Figure 6.3. Note that the effect of your changes is only shown if you e.g. hit Enter or click on a different property. If you accidently click outside of the text object an in the design pane, you'll end up switching the property editor to the window itself, which you don't want. (You can simply click back inside the text object to return the property editor to the text object's settings.)[11]

When you are done (editing properties) – click the Save icon. If this is a *GUI* that you have not previously created or previously assigned a filename to, you'll get a Save As dialogue box. At this point, **MATLAB** is going to save the window design with a .fig extension.

[8] Note that this is to facilitate the positioning of the icon rather than being anything about guns and shooting at the coders behind **Windows**.

[9] I didn't actually read this anywhere – the operation of the editor or Windows has the same feel and intuitive usage as the sort (hopefully) of Windows you you are going to create in your *GUI*(s)).

[10] In reality: **MATLAB** is secretely writing the relevant code and setting the parameter value ...

[11] Unfortunately, the title of the property editor window is completely unhelpful – matlab.ui.control.UIControl when the text object properties are being edited, and matlab.ui.Figure when the (figure) window properties are being edited. So maybe watch out for Figure appearing in the title bar as an indicator or quite what is being edited.

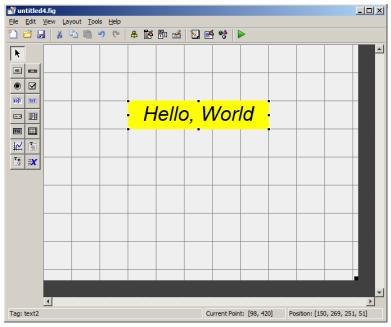

Figure 6.3: Design of the Hello, World window!

Something a little scary now happens – **MATLAB** opens up the code editor window and there is a whole bunch of code in it (a series of *functions* in fact). Note that the code file has a filename the same as you entered in for the .fig window but now with a .m extension (and so is presumably directly associated with the figure you just created). There is nothing we need worry about ... yet. In fact, half the file is taken up with a main *function* that has the comment: `DO NOT EDIT`. Please take this advice ... :o)

In fact, you get given the framework code for 3 functions (pay attention to where each *function* 'ends' ... it is not super-clear ... and a common cause of issues/bugs is accidently placing code within one of these (particularly 1st and 3rd) *functions*:

1. The long one at the top (which you DO NOT edit ...):

   ```
   function varargout = FUNCTIONNAME(varargin)
   % UNTITLED MATLAB code for FUNCTIONNAME.fig
   ```

   `FUNCTIONNAME` – defines the function for your app/program.
   This section ends:

   ```
   % End initialization code - DO NOT EDIT
   ```

   (and you should not place any code in this section).

2. The middle one ...

   ```
   % -- Executes just before untitled is made visible.
   function FUNCTIONNAME_OpeningFcn(hObject, eventdata,
   handles, varargin)
   ```

   `FUNCTIONNAME_OpeningFcn` – allows you to execute any code before the window appears. Such code is typically associated with initialization (setting sup arrays and defining parameters etc.)
   This one ends:

   ```
   % UIWAIT makes untitled wait for user response (see
   UIRESUME)
   % uiwait(handles.figure1);
   ```

   (This is the only one of the 3 that you might place code in (for the purpose of this class).)

3. And ... the third and final one:

   ```
   % -- Outputs from this function are returned to the
   command line.
   function varargout = FUNCTIONNAME_OutputFcn(hObject,
   eventdata, handles)
   ```

   simply allows you to set any output (*function* return) variables that you wish to pass back to the command line. (You need not pass anything back.)
   and ends:

 str='do you like bananas?';

```
% Get default command line output from handles
structure
varargout1 = handles.output;
```

Although there looks like a lot of stuff here, the code is automatically generated and generic and there are both a bunch of blank lines that bloat it all up and lots of comment lines, mostly briefly describing the functions and their inputs (and of which we do not care very much about).

Close the design window (and the code editor if it distracts you). At the command line, type the filename (no extension) to run the automatically generated code **m-file**. A window opens up ... the contents should come as no surprise, because you have just specified them (via the **GUIDE** *GUI* design tool). Your first *GUI*-based program! But one you might notice does not actually do anything – it just sits there unresponsive. Although you can at least close it (because it is automatically generated with the usual basic close/minimize icons plus the name of the **m-file** in the *titlebar*.

6.1.2 *Simple GUI responses [Push Button]*

A *GUI* is only of any real use if it allows some response to input.[12] This is going to involve a little code of your own ... so we'll start with the simplest possible action – a *button* that performs a simple action (in this example – closes the window).

[12] A windows could ne a little error message or warning window, but even then, they typically have an 'OK' button to close them)

Re-run the `guide` program and open up a new window editor (by clicking OK in the **GUIDE** Quick Start window). Now find the Push Button icon, click it, and drag out a push button object in the design pane. You should see a box (with a pseudo 3D shading at the edges) with the text Push Button in the centre as per Figure 6.4. As before, you can edit the properties of the push button object (because the default properties are totally boring) by double-clicking it. Start by editing the font (size) and message. Perhaps 'Go away!'. And then save it.

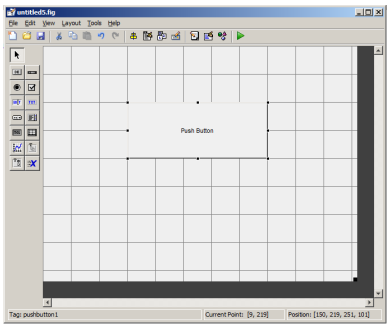

Figure 6.4: Design window with a default push button object.

When it saves, **MATLAB** again opens up the code associated with the figure window that it has automatically generated. There is slightly more code in the file this time and shortly, you'll need to look at it. But for now: ignore it again and type the name of your **m-file** file at the command line. Again, you'll get a window opening with the push button you created in it. Click on it. It does seems to 'respond' (pretends to depress by means of changing the edges with the pseudo 3-D shading) to the mouse click, but ... nothing else happens. This is where YOU (and your amazing coding skills) now come in.

If you have closed the design window, re-run **GUIDE** (» `guide`) and rather than creating a new *GUI* – switch to the Open Existing GUI tab and double-click your filename (of the push button *GUI*) or select and OK. Double-click on the push button object to open up the property editor. We'll make only one (more) change here – down the list of properties, find: 'Tag'. This is the name (ID or *handle*) of the push button object.[13] By default, the name is pushbutton1. Edit this to ... goawayButton (or pick an alternative name) and re-save the GUI.

[13] In essence, no different from a file-name – a unique identifier for an object (/file).

Go to the code editor for the associated m-file (which will have the same name as the .fig figure, remember). In the file we have:

- The main function which we can ignore (and indeed apparently should not be edited!). But for completeness, it consists of:
  - The function definition header line:
    ```
    function varargout = NAME(varargin)
    ```
    where NAME is the name you assigned the file.
  - Some comment lines:
    ```
    % NAME MATLAB code for NAME.fig
    % NAME, by itself, creates a new NAME or raises
    the existing singleton*.
    ```

 str='do you like bananas?';

Note that there is a continuous block of comment (%) lines. **MATLAB** treats this as the `help` text on function `NAME`.

- Then some more, but separated (by blank lines) comment lines.
- Then the body of the function, starting with:

```
% Begin initialization code - DO NOT EDIT
```

and then ending with:

```
% End initialization code - DO NOT EDIT
```

- `function NAME_OpeningFcn` which is executed when the GUI is started up. This is the place to put code for initializing models or whatever (hence the automatically generated part of the function name – `OpeningFcn`).

It is not obvious (to me) what either:

```
% Choose default command line output for NAME
handles.output = hObject;
```

or

```
% Update handles structure
guidata(hObject, handles);
```

actually do ... so ignore these lines for now.
If you need to execute any code when the program/app first runs, place it after these lines.

- There follows another function call:

```
% -- Executes just before NAME is made visible.)
function NAME_OpeningFcn(hObject, eventdata, handles,
varargin)
```

which seems to prepare any data that you wish to return from the main function and back to the command line.
Textbooks helpfully say to ignore this. Great idea.

- Finally, there is:

```
function goawayButton_Callback(hObject, eventdata,
handles)
```

This *function* is executed when your 'Go Away!' push button is pressed. Anything you wish to 'happen', in terms of code executed, when you click on this particular button, goes here in this function.

Note that **MATLAB** seems to be too lazy itself to formally end any of the functions with end. Don't get confused as to where to place code – assume that where you see the next function definition starting, that the previous function has ended. If it helps – add in lines (with end) to end each *function*. Or perhaps add some ASCII art/comment line before each new function to help make it clearer, e.g.

```
% === END FUNCTION ========================================
```

In the current example, after:

```
% -- Executes on button press in goawayButton.
function goawayButton_Callback(hObject, eventdata,
handles)
% hObject handle to pushbutton1 (see GCBO)
% eventdata reserved - to be defined in a future
version of MATLAB
% handles structure with handles and user data (see
GUIDATA)
```

is a blank line (and then the end of the page). This is where any code associated with the *function* `goawayButton_Callback` should go.

In this simple *GUI*, we have only one figure and it is *active* (it has the mouses' attention)[14], so we could simply use the `close` command (*'deletes the current figure'*) on its own (just this single command on its own, on one line).

Insert this simple command (`close`) in the

```
function goawayButton_Callback
```

function, after the last comment line.[15] Save the **m-file** and re-run. Now if you run your program and click on the 'Go Away!' push button, the window does indeed go away (aka, closes).

Phew! So, to recap – you have created a program with a window, and within that window a Push Button object. In the design window, you gave that button a special property, such that when clicked, a message (an *event*) would be passed back to your program. The code (a *function*) that responds (is called) when the button is clicked was automatically generated for you, but with no code inside. You added the code (to close the program/window). And it worked!(?)

[14] Often in operating systems – the active window, i.e. the one that is the one to respond to mouse clicks or key presses, has its titlebar highlighted in a bright color (while inactive ones might be grey.)

[15] Note that automatically generated **MATLAB** code does not seem to ever formally `end` a function as one really should do ...

 str='do you like bananas?';

6.1.3 *Updating object properties (do you like bananas?)*

Bananas. Do you like them? Perhaps the *GUI* can provide an answer (rather than just text statements written to the command line via `disp` as before).

Now you are going to want to think about the design of the *GUI* a little. Perhaps sketch out a design on paper[16] first.

What we want is for the the *GUI* to display a question ('Do you like bananas?'). There will be two options, 'Yes' and 'No' that can be clicked. Depending on which one is clicked, some appropriately supportive, or otherwise, message will appear in response. We need:

1. A plain (static) *text box* as before to display the question.
2. A pair of *push buttons* (again as before ... but 2 of them, rather than just 1).
3. Another plain (static) *text box* to display the answer/response.

And ... we are going to need some code that, depending on which button is pushed, leads to a different message being displayed.

The latter part is not as bad as it sounds. We could have no text initially in the 2nd (static) *text box*. We just need to change its text property (i.e. change the initial no text, to the text of our message). This is mostly a case of working out and using the unique identifier of this text box object AND the identifier of the text property (of the text box object). i.e. you need two bits of information – the ID of the text box, and the ID of the property of the box that sets the actual text to be displayed. You'll see how this pans out shortly.

OK ...

Firstly – re-run **GUIDE** (» `guide`). Create a new *GUI* window with the 4 elements (2 static text boxes and 2 push buttons). It is up to you how you arrange these 4 objects in the design pane. You might be guided how windows in programs that you have used, are designed. At the minimum, it is standard practice to place a 'No' push button next to and aligned horizontally with, the 'Yes' (and often 'Yes' to the right of 'No').

No idiot would design anything like Figure 6.5 and certainly not with those color choices ... but you get the idea.

For each of the objects (2 text boxes and 2 push buttons), rename them (the Tag property) to something more memorable than e.g. button or box, or #1, #2, #3, etc etc..

The code that **MATLAB** generates for bananas.m (my name choice! it need not be yours ...) is not a lot more involved than before. Primarily, there is just a second *function* associated with a mouse click on the 2nd push button. The end of the automatically generated **MATLAB** code now looks like:

[16] The white flat stuff that you write on. Maybe you have forgotten what it is? Clue: it is not an app on iTunes.

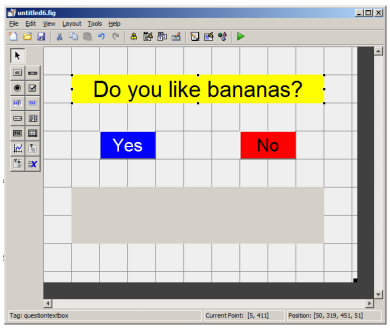

Figure 6.5: (completely) Bananas design window.

```
% -- Executes on button press in yesbutton.
function yesbutton_Callback(hObject, eventdata,
handles)
...
...
% -- Executes on button press in nobutton.
function nobutton_Callback(hObject, eventdata, handles)
...
...
```

The logic is going to be very simple. In fact, we don't need any, because if the Yes button is clicked, **MATLAB** will call the first *function* (my name: `function yesbutton_Callback`), and if the No button is clicked, the second *function* (`function nobutton_Callback`) is called. As alluded to above, how do we get the text to change in the 2nd text box (from the default of no text)?

Unfortunately, **MATLAB** does get all weird here.[17] If you had a friend called Luna, you might reasonably communicate with them via the name 'Luna'. **MATLAB** doesn't do it this way and instead assigns a numeric ID. Think of it as rather than storing information in a database about Luna and by name, information might be stored by SSN instead. So to retrieve or write information about Luna, you do it via their SSN.[18] Here, we want to change a property (the displayed text of the 2nd text box), and you are going to have to get its ID first in order to do it.

First off, you can get the ID of the object property using the `findobj` *function* and assign the result to some memorably variable[19], e.g.

```
h_answertext = findobj('Tag','answertextbox');
```

This is as simple(!) as asking to find the ID of the object which has a `Tag` with value 'answertextbox' (which was the value I set in the design editor).[20]

Put this line of code at the start (after the *function* definition, and after the initial comment lines) of the *function* `yesbutton_Callback` and because you have two buttons, both of which will need to be able to change the text in the 2nd text box – also duplicate this line of code in the *function* `nobutton_Callback`.

Now – we have the ID of the 2nd text box and we can now set its property (from containing no text, to displaying a suitable message). Let us first implement an answer if the Yes push button is clicked. The command to set a property is ... `set`. In our example, the *handle* of the text box we have already obtained and assigned to the variable `h_answertext`. The name of the property we want to change (refer to the column list in the property editor if you like as a reminder) is `'String'`. And the text ... well, you can have whatever you want.

[17] Actually, no weirder than **Python** ...

[18] Or you could think about university student databases and access via the unique student number.

[19] Here, the `h` bit stands for 'handle' but you might chose `id` for ID instead?.

[20] What we might refer to as an ID, **MATLAB** calls a *handle*. Hence commonly an 'h' might appear at the start of a variable name to indicate it contains a *handle*.

`set`
Sets ... the property value of an object. The syntax is:

```
set(h,name,value)
```

where `h` is the handle (the ID obtained via `findobj`), `name`, is the name of a property, and `value`, the value of a property.

 str='do you like bananas?';

The line is then:

```
set(h_answertext,'String','Yes, it is an excellent
fruit.');
```

The complete(!) 2-line piece of code in `function yesbutton_Callback` – finding the ID of the text box and then setting the text property using that ID – should then look like:

```
h_answertext = findobj('Tag','answertextbox');
set(h_answertext,'String','Yes, it is an excellent
fruit.');
```

Save and run the program. You could see something like the result shown in Figure 6.6 (if you click on Yes).

Now extend your program so that an alternative answer is provided if the No button is clicked. This is going to pretty much well be a duplication of the code you have already written for the Yes button.

Other embellishments you could make might be to make the color of the button you clicked change. This is simply a matter of finding its object ID, and setting the property BackgroundColor.[21]

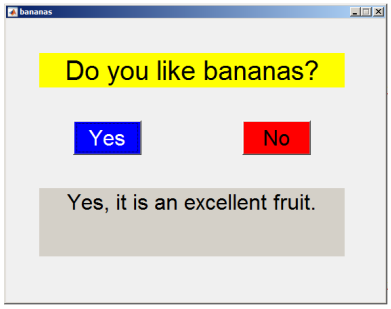

Figure 6.6: (completely) Bananas GUI in action.

[21] For example:
`set(h_answertext,'Backgroundcolor','g');`

To put your coding skills more to the test: how about displaying a 3rd message ('Make up your mind!'?) if someone changes their mind – i.e. if a second button is pressed (after the first one). You'll need a *variable* to remember whether a button (any of them) has already been pressed. You can assign this variable an initial value of *false* (because when the program starts, no button has yet been pressed):

```
var_pressed = false;
```

The idea is whenever either button is pressed, `var_pressed` will become (will be set to) *true*. So before displaying the message in either of the button press *callback* functions, the value of `var_pressed` needs to be tested – a *false* means this is the first time any button has been pressed. Once that initial message is displayed, the `var_pressed` becomes *true*, and when the next time a button is pressed and the value of `var_pressed` tested, a *true* leads to a different message.

All that is needed is:

- A `if var_pressed ... else ... end` structure in each click button callback *function*, to determine which message to show.
- A line initializing `var_pressed` to *false*, which will go in the *function*: `function bananas_OpeningFcn`).

Go add these bits of code.

There is just one problem ...

If you remember – *variables* in *functions* are 'secret' (*private*) and limited (in *scope*) to just that *function*. So the *variable* `var_pressed` which you initialized at the end of `function bananas_OpeningFcn` cannot be 'seen' by the callback *function*.

We can enforce that the same *variable* is seen by multiple *functions* by stating that it is *global* (in *scope*):

```
global var_pressed;
```

This line needs to appear at the start of each *function* in which you need to read or write the value of `var_pressed`, which here will be: both callback *functions* as well as the initialization *function*.

What is going on is that in each *function* that a *variable* is defined as being *global* – the value of that *variable* is linked, so that any change made to the value of that *variable* in any *function*, is seen by all the other ones.[22]

[22] Like multi-way **Skype** call, where the global definition is each person connecting to the conference call.

A complete code for the Yes button call box *function* is:

```
% -- Executes on button press in yesbutton.
function yesbutton_Callback(hObject, eventdata,
handles)
% hObject handle to yesbutton (see GCBO)
% eventdata reserved - to be defined in a future
version of MATLAB
% handles structure with handles and user data (see
GUIDATA)
% MY CODE FOLLOWS > > >
% link 'button-pressed?'  (global) variable
global var_pressed;
% get textbox handle (ID)
h_answertext = findobj('Tag','answertextbox');
% test for whether no button has yet been pressed ...
% set textbox properties (with a message) accordingly
if ~var_pressed
    set(h_answertext,'String','Yes, it is an excellent
fruit.');
else
    set(h_answertext,'String','Make up your mind!');
end
% update button-pressed variable
var_pressed = true;
```

and with the code:

```
global var_pressed;
var_pressed = false;
```

appearing in `function bananas_OpeningFcn` (to initialize the value of `var_pressed` to `false`).

Complete all the code in your program (primarily for the 2nd 'no' click-button) and try it out!

 str='do you like bananas?';

6.1.4 *Simple GUI responses [Sliders]*

We can create a variant on the previous program to illustrate numerical input and output, and introduce the *Slider* object.

1. First create a new (blank) *GUI*.

2. Add a *Static Text* box *object* to ask: 'On a scale of 0 to 10, how much do you like bananas?' (replace the default text by editing the *object*).

3. Add a second *Static Text* box *object* to report the value. Create it blank to start with (i.e. delete the default text).

4. Add ... a Slider! Double-click to edit the Slider object.
Firstly, note that there is a Max and Min property – these are the values assigned when the Slider is at it maximum and minimum position, respectively. Since you want a score for 0 to 10 – edit the Max value. There is also a Value property which will be the default value of the Slider when the program/app starts up. (If you want change the default value.) Make any other e.g. cosmetic changes you fancy. Close the editor ('Inspector') window.

5. Save the *GUI*.

In place of the click-button Callback *functions*, you now have:

```
% -- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)
```

(although if you were paying attention earlier, you would have name the Slider something helpful rather than the default of `slider1`). And a last *function* (`slider1_CreateFcn`) that we shall ignore.

This is not so different form the click-button example – when the slider bar is dragged, or the up/down arrows are clicked (and the slider bar moved that way), this *function* is called. It is then up to you (in code) to:

1. Read the value of the Slider.
2. Do something with that value (i.e. display it!)

As before, we need to get the ID of the Slider object, and then read is Value property.

```
h_slider1 = findobj('Tag','slider1');
bananaindex = get(h_slider1,'Value');
```

(which goes in the `slider1_Callback` function).

To set the text in the Static Text box object, as before you need to obtain the ID of the object:

```
h_text2 = findobj('Tag','text2');
```

(here again ... not the greatest of *variable* names ...)

Now simply set the String property of the Static Text box, to the value of the Slider, contained in variable `bananaindex`.[23]

Now ... if the value of the *variable* `bananaindex` goes above 5, make the text box background blue. And if `bananaindex` is below (or equal to) 5, set the color to red.[24] (You'll need an `if ...`, obviously.)

There are various further refinements that you could make, such as when the program/app starts up, you could read the default value of the Slider and update the display (the Static Text box)[25]. Maybe try to do this.

[23] Remember that you cannot display a number directly where a string is required – use `num2str`.

[24] The Static Text box property is called Foregroundcolor. To set, e.g. add the code:

```
set(h_text2,'Foregroundcolor',[1
0 0]);
```

[25] Code going in the *function*, `OpeningFcn`.

```
224  str='do you like bananas?';
```

6.2 *MATLAB apps*

7
Example codes

```
226  str='do you like bananas?';
```

7.1 Chapter 1 codes

7.2 Chapter 2 codes

Section 2.4

Basic code for loading, storing, and plotting, monthly global temperatures

```
% ******************************************************************
% Program to load in 12 monthly temperature data-sets
% plot each monthly global temperature distribution,
% and generate a MATLAB format animation (M)
% ******************************************************************
% close all currently open figure windows
close all;
% START OF MONTHLY LOOP
for month=1:12
  % create filename
  filename = ['temp' num2str(month) '.tsv'];
  % load data and assign to a new array 'slice'
  temp = load(filename);
  % plot data
  pcolor(temp);
  % store frame for animation
  M(month) = getframe;
end
% END OF MONTHLY LOOP
% ******************************************************************


% ******************************************************************
% Program to load in 12 monthly temperature data-sets,
% store the data in a 3D array,
% and plot each monthly global temperature distribution
% in a separate figure window
% ******************************************************************
% close all currently open figure windows
close all;
% START OF MONTHLY LOOP
for month=1:12
  % create filename
  filename = ['temp' num2str(month) '.tsv'];
  % load data and assign to a new array 'slice'
  temp(:,:,month) = load(filename);
  % create new figure window and plot data slice
  figure;
  pcolor(temp(:,:,month));
end
% END OF MONTHLY LOOP
% ******************************************************************
```

 str='do you like bananas?';

Code for creating an avi format animation

```
% ****************************************************************
% Program to load in 12 monthly temperature data-sets
% plot each monthly global temperature distribution,
% and generate an avi format animation
% ****************************************************************
% Prepare the new file.
vidObj = VideoWriter('my_animation.avi');
open(vidObj);
% START OF MONTHLY LOOP
for month=1:12
   filename = ['temp' num2str(month) '.tsv'];
   temp = load(filename);
   pcolor(temp);
   % Write each frame to the file.
   currFrame = getframe;
   writeVideo(vidObj,currFrame);
end
% END OF MONTHLY LOOP
% Close the file.
close(vidObj);
% ****************************************************************
```

Section 2.4

Code for loading and plotting the continental outline

```
%%% code to plot the continental outline %%%
% load data files
% (and assign the contents to short name variables)
co_start = load('continental_outline_start.dat');
co_end = load('continental_outline_end.dat');
co_lat = load('continental_outline_lat.dat');
co_lon = load('continental_outline_lon.dat');
% determine number of line segments
% (either the segment start, or end, vector will do)
n_lines = length(co_start);
% create new figure window and set axes
% (for the global domain)
figure;
axis([-180 +180 -090 +090]);
% 'hold on' ...
hold on;
% LOOP ...  and draw thin line segments
for k = 1:n_lines
   plot(co_lon(co_start(k):co_end(k)), ...
        co_lat(co_start(k):co_end(k)),'k-','LineWidth',0.25);
end
% adjust the plot aspect ratio to be more map-like
set(gca,'PlotBoxAspectRatio',[1.0 0.5 1.0]);
% draw a nice boundary to the map
h = plot([-180 +180],[-90 -90],'k-');
set(h,'LineWidth',1.0);
h = plot([-180 +180],[+90 +90],'k-');
set(h,'LineWidth',1.0);
h = plot([-180 -180],[-90 +90],'k-');
set(h,'LineWidth',1.0);
h = plot([+180 +180],[-90 +90],'k-');
set(h,'LineWidth',1.0);
% 'hold off' (not strictly necessary)
hold off;
% label plot
xlabel('longitude','fontsize',15);
ylabel('latitude','fontsize',15);
title('Continental outline','fontsize',18);
% export graphics (as postscript)
% (or manually save from the figure windows as a jpg)
print -dpsc2 ch3p2p6.ps;
```

 str='do you like bananas?';

7.3 *Chapter 3 codes*

Section 3.1

Code for using `textscan`

```
% === program to load in data using textscan ===
%
% first - open a 'file pipe' (ID: file_id)
% (to pen the file for subsequent data reading)
file_id = fopen('paleo_CO2_data.txt');
% now ...  load in the actual contents,
% with a data format defined by:  '%f %f %f %f'
% (4 columns of floating point numbers)
my_data = textscan(file_id,'%f %f %f %f');
% now close the file!
fclose(file_id);
% BUT, the contents of my_data is a 'cell array' :(
% => convert to a 'normal' array ...
my_data_array = cell2mat(my_data);
% ...  and now the data array looks like before
% and you can go 'do' something useful with it.
%
% === END ===
```

7.4 Chapter 4 codes

Section 4.2

Code for the `maxxx` function

```
function [s_out] = maxxx(v_in)
% maxxx
%
% Takes a (single) vector as input,
% returns the maximum value.
% Determine number of elements in vector
nmax = length(v_in); % Seed temporary (running maximum)
variable
temp_max = v_in(1);
% Loop through all
% but the first element in the vector
for n = 2:nmax,
   if (v_in(n) > temp_max),
      temp_max = v_in(n);
   end
end
% Set function (return) value
s_out = temp_max;
end
```

Bibliography

Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Pocket Books, 1979. ISBN 0-671-46149-4.

Index