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 math in **MATLAB**, and 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. (Excepting the continually broken **MATLAB** postscript rendering.)

1.1 Using the MATLAB software

1.1.1 Starting MATLAB

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

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

1.1.2 The command line

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

```
Academic License
»
```

or in order versions of the software:

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

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

If you are unfamiliar with using command-line driven software ... Don't Panic![3] Nothing bad can happen, regardless of what you do. Well, almost. It is possible to accidently clear **MATLAB**'s memory of the results of calculations and data processing and close plots and graphs before you have saved them, but **MATLAB** remembers all the commands you type, so in theory it is perfectly possible to quickly reproduce anything lost. (Later on we will be placing the sequence of commands into a file (that is saved) and so ultimately, MATLAB should turn out to be mostly fool-proof.)

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

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

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

Well ... not so exciting. But not so disastrous. **MATLAB** simply has no clue what you are talking about, or rather, `anything` is not a 'key word'[4] 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.

[4] 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 learn good programming practice. Which is why we mostly will ignore the eye-candy and little (if any) guidance will be given as to the functionality of the Graphical User Interface (GUI). Look at this as a lesson for the user (to read the Help, textbook, on-line documentation, or simple go Google for an answer[5]).

[5] i.e. Internet fishing

1.1.4 Help(!)

If stuck at any point – press `F1` or click on the question mark icon on the tool-bar, to bring up the indexed and searchable **MATLAB** documentation.[6]

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

```
» help
```

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

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

[7] 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.[8]

[8] 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.

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

In **MATLAB**, a variable is associated a name to make things rather more easy and convenient. The name can be any sequence of characters you like in **MATLAB**, regardless of whether it is a real or fake word, as long as it does not contain numbers or special characters or spaces. So actually, you are only left with continuous sequences of letters (otherwise knows as 'words'). But you can also create a variable name based on 2 (or more) words, separated by an underscore (_). Valid variable names include:

```
A
B
cat
derpyhooves
this_is_boring_stuff
BIG
big[10]
```

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[11].

So far so useless – you need to *assign* something to it. (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 throwback to the days when computers only had $1/10000000^{th}$ of the memory of your iPhone and were slower than half a lemon.

- **Real (floating point)**[12] – A real number can have a non-integer component, e.g. 1.5 or $6.022140857 \times 10^{23}$. Real numbers also

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

[10] Note that MATLAB distinguishes between lower and UPPER case letters in a variable (i.e. `BIG` and `big` would represent two different and distinct variables). I would strongly advise to stick to all lower case, or all upper case, to avoid possible future confusion. (or come up with a naming convention, of whatever sort (e.g. capital first letter), and stick to it.)

[11] 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.

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

come in different precisions in **MATLAB** (also to do with memory allocation and speed), determining not just the number of decimal places that can be represented, but also the maximum size.

- **String (character)** – One or more characters, but now allowing spaces (unlike in the case of naming variables).
- **Logical** – the variable can be `true` or `false`[13] – 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 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).

[13] As opposed to a Trump variable, that can have many different completely alternative states of 'true'.

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, do not attempt to mix up (combine) variables of different types. **MATLAB** is very forgiving when it comes to combining an *integer* and a *real* number in the same calculation, but in other programming languages, this should be avoided. However, even in **MATLAB**, *strings* and *reals* (or *integers*) are very different things.[14] When necessary, different variable types can be converted between (see **Variable Type Conversion** Box).

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

Variable Type Conversion

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

1. converting from a number to a *string* (`s`)
   - `s = 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.

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.[15] **MATLAB** allows you to define and assign a value to a variable all at the same time, and it will kindly work out the correct type based on the value you assign to it. You assign a value using the assignment operator `=`[16]. 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

[15] Partially true. An Alternative Fact of sorts.

[16] 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.

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 ...)

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

MATLAB will also report in the *Workspace* window, the name and value, *type* (unhelpfully called *Class*), etc of all your current variables (just one currently?). Actually, it is not all quite so simple. If you take a look at the *Class* of the variable `A` in the display window – it is listed as *double* (a *real* number) rather than an *integer*. So by default, if **MATLAB** does not know what you really want, it defines `A` as a double precision real number[17].

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.[18] Also lister here 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
A =
10
```

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

```
B = apple
```

and **MATLAB** is far from happy. As it turns out, a sequence of characters can also refer to a *function*[19] in **MATLAB**, and this is what **MATLAB** looks for (i.e. a match to `apple` in the list or 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. To delineate `apple` as a *string*, you need to encase it in (single[20]) quotation marks:

```
B = 'apple'
```

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

```
A = 'banana'
```

has now replaced the real number `10` with the character string **banana** in variable `A`. This is reflected in the updated variable list details given in the *Workspace* window (and a *Class* now listed as `char`).[21]

[17] 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).

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

[19] 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.

[20] Double `""` quotation marks will not work.

[21] 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`).

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[22], i.e.

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

now does not results in anything being echoed to the command line (but the *Workspace* is still updated to reflect this variable assignment).

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

1.2.2 *Numerical expressions*

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

- `exponentiation` — `^` — raises one number of variable to the power of a second, e.g. a^b, a to the power b, which is written in MATLAB as `a^b`.
- `multiplication` — $\times$ — e.g. a$\times$b, written in MATLAB as `a*b`.
- `division` — `/` — (written as you would expect).[23]
- `addition` — `+` — (guess).
- `subtraction` — `-` — again, obvious/intuitive.

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

The order in which the numerical operators are written down is important and will execute them in a specific order (operators higher up the list, executed first), i.e. first `^`, then `*`,`/`, and last `+`,`-`. There is also *negation*, when you change the sign of a variable, and which is executed immediately after exponentiation. The assignment operator (=)[24] comes last. If you are unclear about the order numerical operators are carried out, then place parentheses `()` around the component of the calculation you wish to be carried out first 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:

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

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

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

```
A =
     1
```

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

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

For now – simply keep mind the existence of *relational and logical operators* and what they look like.

1.2.4 Functions (built-in)

MATLAB provides numerous built-in `functions`[25]. These `functions` have specific names assigned to them, so care needs to be take not to give a variable the same name as a function to avoid getting confused further down the road. Giving an exhaustive list (and brief description) is outside the scope of this document[26]. Common `functions` will be progressively introduced as this text progress. Note that in addition to the on-line Help documentation, information on how to use a function and example uses is provided by typing `help` and then the function name (separated by a space) at the command line.

MATLAB also provides several built-in mathematical constants (saving having to define a variable with the appropriate number). This are simply variables that have been already defined and assigned values, but which you cannot change (hence the term 'constant'). For instance, the value of π, is assigned to a built-in variable

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

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

with the name `pi`. You can access (display) its value by typing its name at the command line:

```
» pi
ans =
    3.1416
```

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

1.2.5 Miscellaneous commands

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

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

Note that a useful trick – if you want to re-use a previously used command but don't want to type it in all over again, or want to issue a command very similar to a previously-used one – is to hit the UP arrow key until the command you want appears. This can also be edited (navigate with LEFT and RIGHT arrow keys, and use Delete 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[27], 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`.[28]

[27] An 'alternative' pi?

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

1.3 *Vectors and arrays #1*

So far, your variables have all be what are known as *scalars* – i.e. single numbers (or strings). One of the most powerful things about MATLAB is its ability to represent vectors (1D columns or rows of numbers or strings) and arrays – 2D and higher dimensional regular grids of numbers or strings. (*matrix*[29] is the name commonly given to a 2-D array.)

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

1.3.1 *Creating vectors*

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

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

or with the value comma-separated:

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

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

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

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

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

which gives the maths-speak representation:

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

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.[30]

[30] 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.

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

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[31].
2. Refer to the Workspace window, although this also ends up a total Fail for long vectors.
3. Use the `length` or `size` function (see Box).
4. Refer to the *Workspace window* ... but ... by default, the Size of variables is not one of the displayed columns (instead, it has to be added from Choose Columns right-mouse-button-click menu item).

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

`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.)

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

```
D = B'
```

will turn the vector `B` into one (assigned to the variable `D`) with he same orientation as `C`. [32] You can also use the `transpose` function.

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.

`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).

[32] 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 raw vector. Simples! Lesson over.

1.3.3 *Addressing elements in vectors*

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

Values can be extracted (or read) from a vector by specifying the index (technically, this should be an integer, but MATLAB is pretty 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!)[33]

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). [34]

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 `.'`).

[33] 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**.

[34] 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 tot find out the exact name (and usage syntax). All this, and we have not even gotten on to matrices (2-D arrays) yet ... So, we'll take a diversion to look at some basic plotting techniques that will make sense now that you can create vectors of numbers to plot (and later, important some 'real' data). Unless you have forgotten how to create vectors already ... :(

1.4.1 *Plotting*

First – create yourself a dummy dataset to plot. You are going to need to create yourself a pair of vectors – these can have any values (all numbers though) in them that you like, but perhaps aim for `1` vector with values counting up from `1` to `10` (or similar) – this will form your *x*-axis, and the 2nd column ... whatever you like. [35] 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 int the habit of looking up the `help` text first (also refer to alternative/simplified help provided in this text). The key information that will get you started appears at the very top of the text that `help` returns on `plot`:

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

This tells you that you need to pass to `plot`, your *x*-axis data vector (by its variable name), followed by your y-axis data vector (by its variable name) – comma separated. Do this, and depending on just what or how random your *y*-axis data was, you should end up with something like Figure 1.1 in a window captioned "`Figure 1`".[36]

This ... is easily the least professional plot ever (aside from anything at all 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 example here, the

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

```
» Y = X.^2
```

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

`plot`

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

Its most basic usage is:

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

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

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

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

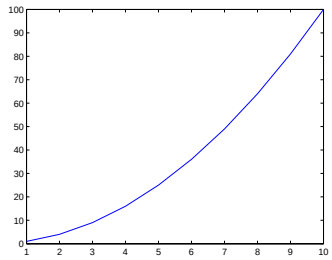

Figure 1.1: Example of the default output of the `plot` function.

default min and max axes limits in fact turn out to be the min and max values in the x and y-axis data because the data is composed of relatively simply/whole numbers. If however the maximum y value was very slightly larger, you'd see that **MATLAB** would adjust the maximum y-axis limit to the next convenient value so as to preserve a relatively simple series of labelled tick marks in the axis scale. In fact, why not try that – replace your maximum data value, with a value that is very slightly larger (an example is given in Figure 1.2). [37] 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).

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

1.4.2 *Graph labelling*

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

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

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

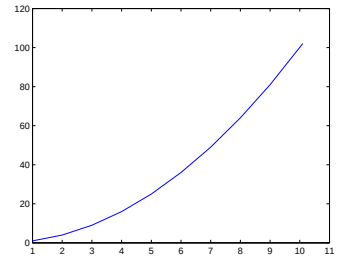

Figure 1.2: A plot illustrating axis auto-scaling (maximum x and y values now slightly larger than 10 and 100, respectively).

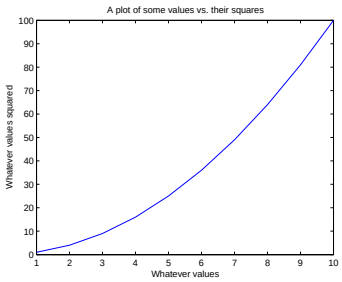

Figure 1.3: A (only very slightly) improved plot.

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

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

1.4.3 *Sub-plots*

You can also have more than one plot in a single Figure window. As an example, create some sine waves using the `sin` function (see `help`)

over the range $0 < x < 2\pi$, e.g.:

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

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

To place several different plots on the same figure uses the `subplot` command [38]. 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.

[38] » help subplot

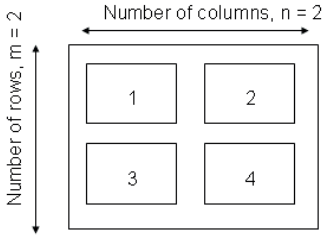

Figure 1.4: Arrangement of subplots.

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 [39]. 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. ;)

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

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

1.5 Vectors and arrays #2

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

1.5.1 Creating matrices and arrays

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

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

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

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

MATLAB then displays the matrix you just entered[40]:

[40] Remember that you can add an `;` to the end would 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
```

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 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)`[41], 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.

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

1.5.2 Basic matrix manipulation

You can treat vectors and matrices (or parts of vectors and matrices), mathematically, as you would treat single values (i.e. *scalars*) but unlike a scalar, the transformation is applied to all specified elements of the array. This applies for all the basic numerical operators[42]. For example, for vector `B` in the earlier example,

```
» 2*B
ans =
    0 2 3 4 5
```

and

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

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

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

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

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

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

Question: What is the sum of the 2nd row of `C`? For a matrix (rather than a vector) as input, `sum` returns the individual sums of each column, and so on its own;

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

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

This is where the `transpose` function (`'`) comes in handy (see earlier). In this case, it flips a (2D) matrix around its leading diagonal (columns become rows, and rows, columns)[43] .

[43] 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.

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

(transposing the matrix turns the rows into columns)

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.)

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

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

Finally, you could transpose the answer:

```
» sum(C')'
```

```
ans = 663
510
782
833
```

to give you a row vector format that corresponds to the rows of the original matrix `C`. [44]

Finally, if you wanted the sum of *all* the elements in the matrix `C` in the example above, you could sum all the columns to give you a row vector of partial sums, and then sum the elements in the row vector to give you the grant total sum of all the elements. You can do this, either in seperate steps:

```
» D = sum(C);
» E = sum(D);
```

or all in one go:

```
» F = sum(sum(C));
```

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

1.5.3 Some matrix math :(

We will not concern ourselves overly with multiplying vectors and matrices together ... but you should be aware that **MATLAB** can do matrix math. For now, it is worth 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
```

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

(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
```

while for matrix multiplication, $\begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$

```
we would write:
» F = E*E
F =
4 6
8 12
```

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

1.6 Loading and saving data

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

1.6.1 Where am I?

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

MATLAB has a default directory that it starts up in and looks at first. For basic Windoz installations[45], this directory is:

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

So, where the `load` command requires a filename to be passed, you will need to enter either the full location of the file; i.e., starting with the drive letter (e.g. as per displayed in the Windows Filemanger address bar, or the relative path to where the file is located.

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

```
» cd DIRECTORY_PATH
```

where `DIRECTORY_PATH` is the path to the directory in which you want to work from and where you want your data files (and later, code files) to live.

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

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

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

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

```
» load(filename)
```

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

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

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

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

The command `addpath` will add a search path to the MATLAB workspace. e.g.

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

are working from (duplicating the functionality of the `cd` command) and add paths to search (duplicating the functionality of the `addpath` command).

1.6.2 *Loading and importing data*

The simplest way (other than via the **MATLAB** GUI and the beautiful green Import Data icon) is to use the `load` function (see Box)[46].

As a brief exercise and practice using `load` – first download the data file `etheridge_etal_1996.txt` from the course webpage[47]. You might start by viewing the contents of the file by opening it in any text viewer (or **Excel**). This is always a good place to start as it enables you to see what you are getting yourself in to (i.e. format of the file, any potential formatting issues, approximate size and complexity of the dataset, etc). Then import the data into the **MATLAB** workspace using the `load` command. Becasue the data is a plan text (ASCII) format and not a special MATLAB .mat file, you need to specify the format:

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

Try simply 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 (see Box) to provide a crude view of the data. To view the contents of the variable in the Variables window – double click on the name of the variable in the **MATLAB** Workspace window. This should open up a spreadsheet-like window in which the data can be viewed, sorted, and even edited.

For practice – plot the data[48], remembering to label the figure appropriately[49]. If you just type `plot` and pass the (here: default) name of the data array:

```
» plot(etheridge_etal_1996);
```

... strange things happen (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.[50] Instead, if you remember, the format of the **MATLAB** `plot` function is:

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

So you need to specify each column of the data (i.e. each vector) separately. If you recal how to specify an entire row or column of an array, you should see that you need to type:

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

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

[47] `http://www.seao2.info/teaching.html`

[48] using `plot`

[49] 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.

[50] And this is why you should remember to use the Scatter (or (X,Y)) Chart in **Excel** for plotting (x,y) data.

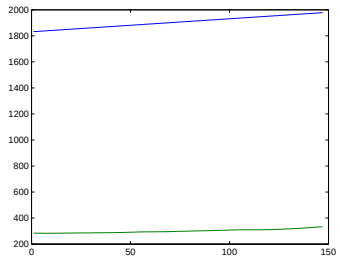

Figure 1.5: Result of simply throwing the entire data matrix at `plot` ....

If this is not obvious ... break it down and create explicit x and y data vector variables, e.g.

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

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

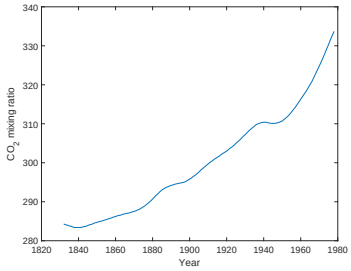

Figure 1.6: Spline fit to measured changes in CO2 concentration in Law Done ice core, following *Etheridge et al.* [1996].

1.6.3 *Saving and exporting data*

Arrays of numbers can be saved in a plain text (ASCII format) by means of the `save` function in a simple reverse of the use of `load` (see Box). Try re-saving the ice-core data as an ASCII format text file (with a new filename) ... and then load it in again.

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

```
» save(filename)
```

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

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

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

1.6.4 *Loading and saving the workspace*

The entire workspace (including all variables and their values, or just the values in a single variable if you wish) can be saved to a file and then later re-opened. The file format is specific to the **MATLAB** program and the file-name extension by default is `.mat`. You might find this very helpful to use in long lab exercise or large modelling projects, particularly if you do not come back to work at the exact same computer each time or wish to use continue the same piece of work on a laptop elsewhere. Try saving the current Workspace, closing down the **MATLAB** program. Re-running it, and then loading in your saved `.mat` file. [51]

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

[51] 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.

1.7 *Basic data processing*

As an example to kick-off some data-processing tricks, load in the Phanerozoic CO_2 dataset: `paleo_CO2_data.txt`. You can just import it into **MATLAB** using the `load` function. However, there is a complication here – unlike the ice core CO2 dataset, you now have 4 columns in the array[52]. The first column is age (Ma), the second the mean CO_2 value, and the 3d and 4th are the low and high, respectively, uncertainty limits. Not forgetting in the space of 5 minutes (I hope!) how to reference specific columns of data in a matrix[53] – plot the mean paleo CO_2 value as a function of age (in Ma). If you closed the previous Figure window (see earlier), it is not essential to explicitly open one (using the `Figure` command) – when you use the `plot` command, if there is no open Figure window, **MATLAB** will kindly open one for you. How thoughtful. The result should be something like 1.7. O dear ...

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

Your options are then:

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

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

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

```
» help sort
```

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

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

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

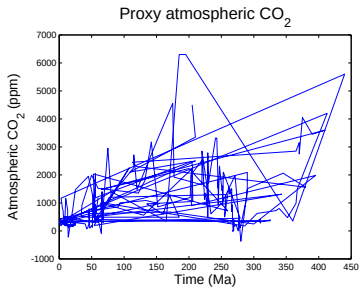

Figure 1.7: proxy reconstructed past variability in atmospheric CO2.

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 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).

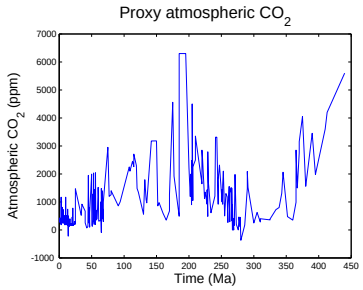

Figure 1.8: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

For another example: download the historical global temperature anomaly dataset: `647_Global_Temperature_Data_File.txt`[54]. The columns are: (1) year, (2) annual mean, and (3) 5 year (running?) mean. Plot (separately) both the annual mean, and the 5-year mean. [55]

The 20th century average global temperature across land and ocean surface areas is apparently 13.9°C. So firstly, try changing the temperature anomaly data into absolute temperatures (by adding the 20th century global average value to all the data values), and then re-plot.

Next, convert the temperature units (of both annual mean and 5-year mean) – from °C to °F. An approximate conversion is:

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

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

Create a new data array in **MATLAB**, with year as the first column (as per the original data) and the 2nd and 3rd columns as annual mean and 5-year mean temperatures in units of °F.[56] If it help – 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).

Re-plot (in **MATLAB**) the final temperature trends in °F.

Analogous to the data processing you have just carried out, in which you applied a scaling and then an offset in order to convert from °C to °F, as a last example to get you familiar with some very basic data processing and array data manipulation, we are going to transform a sediment core $\delta^{18}O$ time-series into an estimated history of glacial-interglacial changes in sea-level. The scientific backstory is ...

Throughout the late Neogene[57], sea level has risen and fallen as continental ice sheets have waned and waxed. The main cause of

[54] https://climate.nasa.gov/vital-signs/global-temperature/

[55] plus labels, title, etc etc

[56] Remember – MATLAB will happily do the same to all values in a vector in an equation, as for a single scalar value in the same equation. Start by creating a new array with a single column, copied form the 1st column of the original array.

[57] 23.03 millions years ago (end of the Oligocene) to present is the Neogene Period in Earth history.

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

Glacial ice has a lower $^{18}O/^{16}O$ isotopic ratio than mean seawater[58]. When ice volume is high, seawater has relatively high $^{18}O/^{16}O$ ratio. When ice volume is low, seawater has relatively low $^{18}O/^{16}O$ ratio. If the average $^{18}O/^{16}O$ ratio of glacial ice is constant with time, then changes in the average $^{18}O/^{16}O$ ratio of seawater will linearly approximate changes in the total volume of ice and by inference, sea-level. We (at least, I am) are interested in all this because knowing how ice volume and sea-level changed over the glacial-interglacial cycles has all sorts of important implications for understanding how climate change (e.g. via ice sheet albedo) and global carbon cycling and atmospheric CO2 (e.g. via changes in the area of exposed continental shelves and carbon stored in soils and above-ground vegetation).

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

To start with we need to reconstruct past changes in the oxygen isotopic composition of the ocean. Handily, the $^{18}O/^{16}O$ ratio of foraminiferal calcite isolated from marine sediments is primarily a function of the $^{18}O/^{16}O$ ratio of the water together with the temperature of the water[59]. By measuring the $^{18}O/^{16}O$ value of calcite downcore we are sampling $^{18}O/^{16}O$ with a progressively older age. In this way we can reconstruct how ocean $^{18}O/^{16}O$ has changed over time. These measurements are reported in units of parts per thousand (‰) and written as $\delta^{18}O$.

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

How to turn (scale) changes in $\delta^{18}O$ into sea-level change? Evidence from dated coral reef terraces suggest that sea-level was around 117 m lower at the peak of the last glacial (ca. 19 ka). We could then assume that the change in $\delta^{18}O$ from modern (preindustrial) to LGM equates to 117 m sea-level change, and hence create a continuous past sea-level curve from all the $\delta^{18}O$ data by applying a simple scaling factor[60]. So:

- You first need the foraminiferal calcite $\delta^{18}O$ data. (Unless you

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

want to go drill a long cylinder of mud from 3000 m down in the Atlantic Ocean, pick out all the microscopic foraminifera of a single species from samples of mud that you have carefully washed, blah blah blah ...) So, from the course web page; download the file `sediment_core_d18O.txt` and save it locally.

- Load this file into **MATLAB**.

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

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

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

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

  Scale the δ^{18}O so that it represents changes in sealevel, relative to modern (0 m)[61].

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

[61] HINT – first determine the difference in δ^{18}O between time zero and 19 ka. This gives you the range of δ^{18}O that maps onto a sea-level change of 117 m. This is pretty much the same as knowing the scaling between °C and °F in the previous example. There is then an offset to apply so as to make the sealevel change at time 0 years, 0 m, but in essence, this is no different from applying the offset previous to turn 0°C to 32°F (admittedly, the analogy is backwards). You also might transform the δ^{18}O data such that it has a value of zero at 0 ka (but retains the original amplitude of variability.

1.8 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`, it to graph the data without the line joining the points. Scrolling a little the way down » `help plot`, it turns out that there are a number of options for color, line style, and marker symbol that you list together as a single parameter, straight after the parameters for x and y vectors. By default, **MATLAB** plots a solid line in blue with no marker points. Obviously, we could forego the sorting and plot a sane graphic (hopefully) by plotting just points and having no line between them. Hell, you could even be radical and use a different color ... Or, you could specify a symbol and no line. The choice of colors is your oyster, as they (almost don't) say. e.g. Figure 1.9.

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

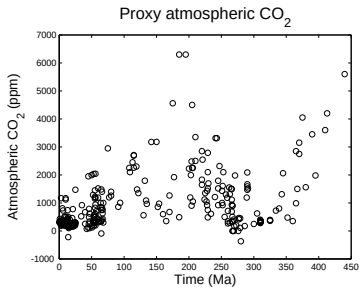

Figure 1.9: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

1.8.2 Plotting multiple data-sets

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

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

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

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

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

the `legend` command. For instance you have managed to successfully plot the mean CO_2 values as discrete black circles, and the minimum and maximum uncertainty limits as blue and red lines, respectively, you could call:

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

and it should end up looking like Figure 1.10.

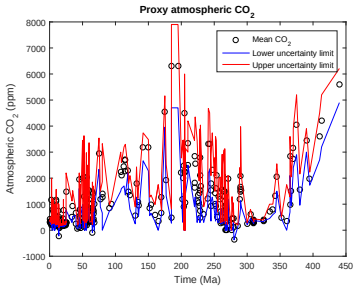

Figure 1.10: Proxy reconstructed past variability in atmospheric CO2 (sorted data).

1.8.3 *Scatter plots*

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

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

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

(where `X` and `Y` are vectors). This almost could not be more straightforward. Make yourself an `X` and `Y` vector out of the loaded-in dataset (or if you are feeling brave, you can pass in directly the appropriate parts of the dataset array), close the existing Figure window[62], and `scatter`-plot the (mean) CO_2 data.

[62] See earlier.

Perhaps a little disappointingly, the default (Figure 1.11) (plus added labels) looks a little like one of the plots before. However, `scatter` can plot color-filled symbols, but more powerfully, can scale the fill color to a 3rd data value (vector), in a sort of pseudo 3D x-y-z plot. For instance, it will be duplicating information that is already presented (y-axis), but you could color-code the points, by the y-value, i.e. the atmospheric CO_2 value. e.g.

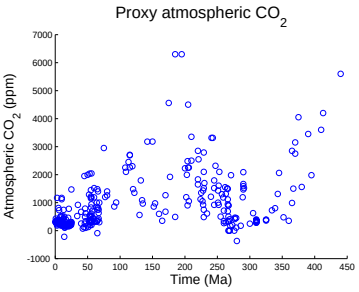

Figure 1.11: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

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

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

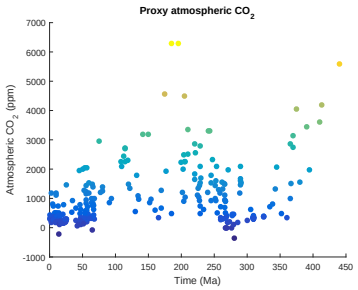

Figure 1.12: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

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

resulting in Figure 1.12.

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

2004 and ending June 30, 2005. The data file can be found on the course webpage (`data_USGS.txt`). The columns are: (1) day, (2) latitude, (3) longitude, (4) depth, and (5) magnitude. Carry out the following:

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

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

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

4. How many earth quakes in total were recorded?[63]

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

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

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

aftershock distribution made people very wary of the (low) early magnitude estimates - the area of dense aftershocks often delineates the part of the fault that ruptured, and scaling laws relate rupture length to magnitude.

Create a figure with multiple panels, showing:

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

1.8.4 Histograms

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

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

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

```
» hist(data_USGS(:,5),M);
```

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

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

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

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

1.8.5 *Simple 2D data and bitmap visualization*

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

First off – as before, view the data in the array viewer, just to get a feel for what you are dealing with here (although you are unlikely to be much wiser after doing so). So go ahead and employ the `pcolor` function in its simplest possible usage (see Box). You can see (Figure 1.13) 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?[64] If you guess right first time (three different basic transformations of a matrix were described), you get Figure 1.14.

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

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

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

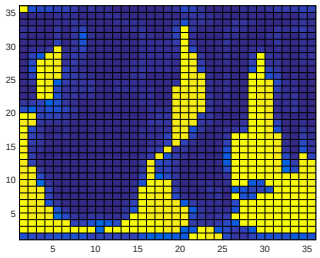

Figure 1.13: A 2D plot of some random gridded model data.

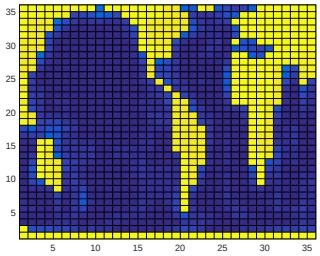

Figure 1.14: A 2D plot of some random gridded model data ... but with the underlying data matrix re-orientated before plotting.

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

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

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