5.1 Further data input

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

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

5.1.1 Formatted text (ASCII) input

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

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

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

opening and closing files

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

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

`textscan`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
» fclose(openfile_id);
```

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

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

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

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

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

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

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

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

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

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

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

```
my_data_array = call2mat(my_data);
```

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

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

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

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

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

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

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

`cell2mat`

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

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

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

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

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

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

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

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

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

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

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

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

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

3. Now close the file:

```
fclose(openfile_id);
```

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

```
my_data_array = call2mat(my_data);
```

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

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

5.1.2 *Importing ... Excel spreadsheets*

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

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

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

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

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

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

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

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

```
» A(4,3)
```

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

```
» C(4,3)
```

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

```
» C{4,3}
```

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

`xlsread`

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

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

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

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

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

5.1.3 *Importing ... netCDF format data*

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

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

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

1. Open the netCDF file by

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[18] Your choice of 2D plotting function.

[19] See subsequent section.

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

other refinements that help visualise the data, include:

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

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

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

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

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

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

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

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

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

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

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