6
Further ... Programming

In this chapter we'll get some (more) practice building programs and crafting (often) bite-sized chunks of code that solve a specific, normally computational or numerical (rather than scientific) problem (*algorithms*) [1].

[1] According to the all-mighty Wikipdeia (and who am I to argue?) – an "algorithm ... is a self-contained step-by-step set of operations to be performed. Algorithms perform calculation, data processing, and/or automated reasoning tasks."

6.1 *find!*

So – a single **MATLAB** function gets a high-level section, all to itself. Either it's really powerful and useful, or I am running out of ideas for the text[2].

[2] It is really powerful and useful.

find ... finds where-ever in an array, a specific condition is met. If the specific condition occurs once, a single array location is returned. The specific condition could occur multiple times, in which case find will report back multiple positions in the array.

find
MATLAB defines find, with a basic syntax of:

```
k = find(X)
```

as 'return[ing] a vector containing the linear indices of each nonzero element in array X'. That means ... nothing to me. This is going to have to be a job for some Examples ... (in order to see what find is all about).

What do I mean by a 'specific condition'? Basically – exactly as per in the if ... construction – a conditional statement being evaluated to true.

OK – some initial Examples.

Lets say that you have a vector of numbers, e.g.:

```
A = [3 7 5 1 9 7 4 2];
```

and you want to find the maximum value in the vector – easy[3]

[3] I hope so ... check back earlier in the course on max.

But ... you want to find *where* in the vector the maximum value occurs. Why might you want to do this? Rarely do you have a single vector of data on its own – generally it is always linked to at least one other vector (often time or length in scientific examples). Trivially, our second vector might be:

```
B = [0:7];
```

and is time in ms. The question then becomes: at what time did the maximum value occur? Obviously, this is easy by eye with just 8 numbers, but if you had 1000s ...

We can start by determining the maximum value.

```
c = max(A);
```

Now, we use find to evaluate where in the array A (here: a vector) the element with a value of max(A) occurs, or where the condition d == c is true,where d is the element in question (the maximum value). So:

```
find(A(:)  == c);
```

should do it. Here, what we are saying is: take all of the elements in A and find where an element occurs that is equal to c (the maximum value which we already determined). Try it, and MATLAB should return 5 – the 5th element in the vector.

Finally, if we assign the result of find to d, we can then use d to determine the time at which the value of 9 occurred, i.e. B(d) which evaluates to 4 (ms):

In this example, find returned just a single element, but if we instead had:

```
A = [3 9 5 1 9 7 4 2];
```

The maximum value is still the same (9) but now ...

```
» find(A(:)  == c)
ans =
    2
    5
```

What has happened is that `find` has determined that there are 2 elements in vector `A` that satisfy the condition of being equal to `c` (9) and these lie at positions (index) `2` and `5`. The result vector, if you assigned it to the variable `d` again, can be used just as before to access the corresponding times in vector `B`;

```
» d = find(A(:)  == c); » B(d)
ans =
    1 4
```

i.e. that the times at which the values of `9` occur are `1` and `4` (ms).

Any of the relational operators (that evaluate to *true* or *false*) can be used. In fact – looking at it this way leads us to maybe understand the **MATLAB help** text, because *true* and false*a* are equivalent to `1` and `0`, and `find` is defined as a function that returns the indices of the non-zero elements in a vector. By writing `A(:)  == c` we are in effect creating a vector of `1`s and `0`s depending on whether the equality is *true* or not for each element. You can pick apart what is going on and see that this is the case, by typing:

```
» A(:)  == c
ans =
    0
    1
    0
    0
    1
    0
    0
    0
```

(the statement being *true* at positions (index) `2` and `5`, which is exactly what `find` told you).

For instance, we could ask `find` to tell us which elements of A have a value greater than 5:

```
» find(A(:)  > 5)
ans =
    2
    5
    6
```

(Inspect the contents of vector `A` and satisfy yourself that this is the case.)

We can also use `find` to filter data. Perhaps you do not want values over 5 in the dataset. Perhaps this is above the maximum reliable range of the instrument that generated them. Having obtained a vector of locations of these values, e.g.

```
d = find(A(:)  > 5);
```

we can plug this vector back into `A` and assign arrays of zero size to these locations – effectively, deleting the locations in the array, i.e.

```
A(d) = [];
```

They it, and note that the size[4] of `A` has shrunk to `5` – all the other elements remain, and in order, but the elements with a value greater than 5 have gone. You could apply an identical deletion (filtering) to the time array (`B(d) = []`).

[4] Use the command `length` or view in the **Workspace** Window.

Play about with some other relational operators and criteria, and make up some vectors of your own until you are comfortable with using `find`.

Back to the 'quake Example: Find[5] how may earth quakes there were bigger than M = 8? Also determine how many quakes occurred bigger than M = 7, 6, 5, 4, and 3. Determine the day on which the magnitude 8.7 shock occurred.

[5] Intentional joke *and* clue.

In the first problem (number of quakes greater than a specified limit) – you need ask find to return the row numbers for all quakes satisfying the condition: `magnitude > 8.0`. `find` will return you a column vector. You don't actually need to worry about or access the contents of the vector, you just need to know how many elements there are in the vector (because there will be one element for each occurrence of `magnitude > 8.0`). This is the same as its `length` (see earlier and/or **help**).

In the second problem – you need to find the row number of the quake magnitude data which satisfies the condition: `magnitude > 8.7`. Knowing the row number, you can then access the data column containing the sate information, and hence extract the day and solve the problem.

All these problems can actually be solved in a single line of **MATLAB**, but feel free to break it down into multiple steps.

In the sealevel (oxygen isotope) Example, you could start by determining the maximum and minimum sea-levels that have occurred over the last 782,000 years. Then ... because it would be helpful to know *when* the minimum and maximum sea-level heights occurred, use the `find` function to find the data row in which the minimum and maximum values occurred. Once you know the respective data rows, you can then easily pull out the ages.[6] Find the ages of both minimum and maximum values.

Also find all the occasions (times) on which sealevel was higher than today (modern). (Or equivalently, when the oxygen isotope value, that we are assuming directly reflects changes in level, was lower than modern[7].)

You can also ask questions based in time, such as what was the sealevel (or oxygen isotope value) at 21 ka (i.e. without having to look through the data manually and determine on which row 21 ka occurs, because this is exactly what `find` can do this for you)? This can be particularly useful if the value of time is calculated or passed in from elsewhere, rather than specified as e.g. 21 ka, because you may not *a priori* know what the value will be, hence automating the script with `find` is super useful. Effectively then you are creating an *algorithm* for taking a time input and determining sealevel.

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

[7] Lower d18O => less ice volume => higher sealevel.

For an Example of data-filtering – dig out the paleo-proxy (not ice-core) atmospheric CO_2 data you downloaded. One further way of plotting with `scatter` is to scale the point size by a data value. We could do with by:

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

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

This leads us to a new use for `find` and some basic data filtering. The simplest thing you could do to ensure no zero values, would be to add a very small number to all the values. This would defeat the 'no zero' parameter restriction, but would not help if there were negative values and you have now slightly modified and distorted the data which is not very scientific. Substituting a `NaN` for problem values is a useful trick, as MATLAB will simply ignore and not attempt to plot such values.

`NaN`
... is **Not-a-Number** and is a representation for something that cannot be represented as a number, although if you try and divide something by zero **MATLAB** reports `Inf` rather than a `NaN`.
`NaN` can also be used as a function to generate arrays of `NaN`s. The most common/usage in this context is:

```
N = NaN(sz1,...,szN)
```

which will (according to **help**) "generate a a sz1-by-...-by-szN array of NaN values where sz1,...,szN indicates the size of each dimension. For example, `NaN(3,4)` returns a 3-by-4 array of NaN values."

So first, lets replace any zero in the CO_2 column of the data with a `NaN`. The compact version of the command you need is:

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

But as ever – perhaps break this down into separate steps and use additional arrays to store the results of intermediate steps, if it makes it easier to understand, e.g.

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

What this is saying is: first find all the locations (rows) in the 2nd column of `data` which are equivalent (==) to zero. Set the CO_2 value in all these rows, to a `NaN` (technically speaking: assign a value of `NaN` to these locations). You have now filtered out zeros, and replaced the offending values with a `NaN` and when **MATLAB** encounters NaNs in plotting – it ignores them and omits that row of data from the plot.

Alternatively, we could have simply deleted the entire row containing each offending zero. Breaking it down, this is similar to before in that you start by identifying the row numbers of were zeros appear in the 2nd column, but now we set the entire row to be 'empty', represented by `[]`:

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

If you check the **Workspace window**[8], you should notice that the size of the array data has been reduced (by 4 rows, which was the number of times a zero appeared in the 2nd column).

[8] Or:
```
» size(data)
```

We are almost there with this example except it turns out that there is a CO_2 proxy data value less than zero(!!!) We can filter this out, just as for zeros. I'll leave this as an exercise for you[9] ... The plot should end up looking like Figure 6.1. As another lesson-ette, given that the circles are insanely large ... try plotting this with proportionally smaller circles[10].

[9] But you might e.g. use <=.

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

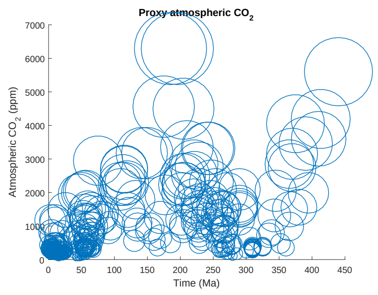

Figure 6.1: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).

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

It turns out this is not entirely trivial because as you have seen, the data is not as well behaved as you might have hoped. In fact, it is just like real data you might encounter all the time! Before you do anything – break down into small steps what you need to do with the data, as this will inform what (if any) additional processing you might have to carry out on the data. It should be obvious, that to create a CO_2 difference, *relative* to the mean, you are going to have to divide by the mean value (column #2 in the array). So first off –

if any of the mean values are zero, it is all going to go pear-shaped. Actually, equally unhelpful, or at least, lacking in any meaning, may be negative values. If you inspect the data (in the **Variable window**), there are both zeros and negative values for mean CO_2 proxy estimates. We need to get rid of these. Follow the steps as before. You may also have to process the min and max values should they turn out to be the same. Likely you are going to have to delete all the rows in which (1) column #2 values are zero or below, and (2) column #3 and #4 values are equal (you could also try the `NaN` substitution and see if it works out). (If you need a slight hint ... one possible answer is here[11] , but try and work it out for yourself.)

[11] In this possible solution – all rows in the array `data`, with mean CO2 values less than or equal to zero, are deleted. Also, all rows for which the max and min values are the same, are also deleted.

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

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

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

It should end up looking something like Figure 6.2 in which you can see the high relative uncertainty (bight colors) prevail at low CO_2 values and 'deeper time' (ca. 200-300 Ma). The colorbar title (label) is maybe not ideal, nicer would be one aligned vertically rather than horizontally. We'll worry about that sort of refinement another time.

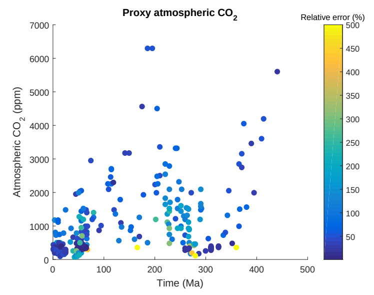

Figure 6.2: Proxy reconstructed past variability in atmospheric CO2 (scatter plot).