Matlab bootcamp – Class 4

Written by Kyla Drushka

More on curve fitting: GUIs

Thanks to Anna (I think!) for showing me this.
A very simple way to fit a function to your data is to use the **Basic Fitting GUI**. First plot the data; then, in the Figure window, go to the menu **Tools ➔ Basic Fitting**. This allows you to instantly fit polynomial functions of different orders (e.g. linear, quadratic, cubic...) as well as interpolate. You can then export the coefficients from the fit, and also to save the code that Matlab used to generate the figure. To save the code, first use the fitting GUI to fit the line, and then go back to the figure window and select the menu item **File ➔ Generate Code.**

There is also a more in-depth GUI called the **Curve Fitting Toolbox**. To run this toolbox, type the command

```
>> cftool
```

Again, you can manipulate the way you fit to your data, and also save the code that is Matlab uses for the fit using the menu item **File ➔ Generate Code** in the curve fitting window. Note, however, that the student version does not come with the curve fitting toolbox.

Other variable types: structures and cells arrays

If you have data that contain a variety of sizes and/or file-types, it can be limiting and frustrating to use matrices. There are a couple of other formats that allow for much more flexibility.

Cell arrays

A cell array is a type of variable with "containers" called cells. Each cell can contain any type of data (e.g. a vector or matrix, a string, or even another cell). Cell arrays are useful, for example, when you have data of different sizes.

Create a cell array the same way you create a matrix, but use curly braces `{}` around the array, and remember that you can insert data of any size/format in each cell. As with matrices, cells on the same row are separated with commas, and you use a semicolon to indicate a new row of cells. E.g.

```
>> newarray={1:5 , 'a string' , 'another string' ; [] ,
ones(4,4) , 0}
```

`newarray` is a 2x3 cell containing data of different formats and sizes in each cell.
You can also create cell arrays of any size using the command **`cell`**, e.g.

```
>> myarray=cell(4,3,2)
```

Arrays are indexed using curly braces, allowing you to access data in cells. For example, to access the cell in the first row and second column of `newarray`:

```
>> newarray{1,2}

ans =

a string
```

You can also modify arrays this way. For example:

```
>> newarray{1,3}=99
```

Structure arrays

Whereas cell arrays contains data in cells that are indexed numerically, **structure arrays** store data in *fields* that you access by name. The fields can contain any data type. E.g. create a new structure containing data from a mooring that contains two instruments:

```
>> moor(1).datatype = 'wind speed'
>> moor(1).instrument_name = 'anemometer';
>> moor(1).units = 'm/s';
>> moor(1).time = 1:10;
>> moor(1).data = [2 3 0 11 2.2 5 3 NaN 8 10];

>> moor(2).datatype = 'SST'
>> moor(2).instrument_name = 'CTD';
>> moor(2).units = 'degC';
>> moor(2).time = [2 3 4 5];
>> moor(2).data = [28.5 28.7 29.2];
```

In this example, we have specified a few parameters for each instrument in different *fields*. The fields are datatype, instrument_name, units, time, and data.

Reading and storing data (part 2)

Previously, we talked about saving data as .mat (Matlab-format) files. Now let's talk about some other options, and some other file-types you might encounter. Note: there is an exhaustive list on the Matlab website: http://www.mathworks.com/help/matlab/import_export/supported-file-formats.html
I will only talk about a couple of options, but I suggest checking out this site!

Definitions

Delimiter – one or more characters used to define the boundary between adjacent values (commonly used delimeters are commas, semicolons, spaces, or tabs)

ASCII data – text data that is encoded using ASCII, in which the numbers from 0 to 127 each correspond to a character in the alphabet or a numbers).

Pros: easy to read – e.g. it can be opened in a text editor.

Cons: large file sizes. Each character requires 1 byte per character (e.g. the number 12345678.9012345 has 16 characters, so would require 16 bytes).

Binary data – data encoded in binary.

Pros: smaller file sizes: a binary-encoded double such as 12345678.9012345 is only 8 bytes.

Cons: there are different ways of encoding binary, and it can be a trick to load them correctly.

CSV (comma separated value) files:

The function csvread reads CSV files (which are an easy and generic format that data can be exported from Excel, for example).

```
>> data=csvread('csv_example.csv');
>> size(data)
```

`csvread` has several options, such as starting at a certain row/column.
Pros: simple, compatible with Excel, you can view your data easily (it's just text)
Cons: if the file contains any text (e.g. a header), it won't work; not very space efficient (potentially large file sizes)

fopen

Often, Matlab has to "open" a file before it can read the data. Once it is open, you can use other commands to actually read the data. Do this using the **`fopen`** command:

```
>> fid=fopen('datafile.txt','r');
```

The first argument (datafile.txt) specifies the file you want to open; the second argument specifies whether you want to read, write, append, etc. (see **`help fopen`** for details) to the file. In this example, we are opening the file datafile.txt and we will only read from it, not write to it. **`fopen`** returns the "id" of the file that it opens; store this "id" in a variable (e.g. `fid`), which you will later refer to when you want to actually read the data. If `fid` has the value -1, that means you could not successfully open the file.

When you are finished with the file, it is important to close it (Matlab will slow down and can crash if you open too many files at once without closing them). Do this using the command `fclose`, e.g.

```
>> fclose(fid)
```

Now that your file is open, you can use some different commands to actually read the data. For example:

`fgetl` – reads data line by line

Pros: very simple and fast. Good for a "first look" at a new dataset. See the help file for a nice example.

Cons: not efficient; you then have to process the data line by line.

`textscan` – for formatted text data ; data returned in a cell array

Pros: very versatile – you can load multiple formats (integers, strings, doubles, etc) separated by different delimeters at one time

Cons: only works for formatted text data.

`fread` – for binary data

Pros: best option for binary files. Good for saving (and then re-reading) your own data to deal with huge file-sizes.

Cons: you need to know some details about the content of the files (e.g. integer data? Singles? Doubles?)

Writing data

You may also want to store your data in formats other than .mat, for example if you want to share it with non-Matlab users. Here are some tools (analogous to the reading tools).

`csvwrite` writes a matrix of data to a CSV file (which can be easily read by Excel, for example). Pros: easy! Cons: limited to 5 significant digits. no header info can be included; you are limited to one matrix. E.g.

```
>> M=randn(100,200);
>> csvwrite('newfilename.csv',M);
```

`dlmwrite` writes a matrix of data to a file using a delimiter of your choice). Pros: easy, more flexibility and precision than `csvwrite` . Cons: no header info can be included; you are limited to one matrix. E.g. (using a tab as the delimiter)

```
>> dlmwrite('mydlmfile.dat',M,'delimiter','\t');
```

`fprintf` writes formatted data to a text file. Pros: flexibility (you can combine text and numerical values). Cons: so many options means that it takes some struggling to figure it out.

`fwrite` writes formatted data to a binary file. Pros: binary files are usually smaller than text. Cons: binary files aren't really "stand-alone", i.e. without information about how they were saved, they may be difficult to read.

NetCDF format

"**NetCDF is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data."**

NetCDF data are often used for sharing climate-related data sets (the NetCDF is strongly supported by UCAR).

NetCDF data-files consist of ***dimensions*** (e.g. time, longitude, depth) and ***variables*** (e.g. wind speed, SST) . *Dimensions* are vectors; *variables* are generally matrices whose size is governed by one or more of the *dimensions*. In addition, NetCDF files usually contain meta-data, called ***attributes***. For example units for the variables and dimensions, flags used for missing values, and even information about who produced the data set and how.

Unfortunately, Matlab does not come with good built-in NetCDF tools. I suggest downloading one of the software toolboxes developed by other groups. A list of these can be found here: http://www.unidata.ucar.edu/software/netcdf/software.html#MATLAB . There are also a number of command-line tools useful with Unix-based systems (e.g. NCO and CDO).

You can still do the basics with the built-in Matlab tools, though, which include the functions
`ncinfo` – `returns information about what is stored in the file`
`ncdisp` – `returns information about a certain variable`
`ncread` – `returns the data for a specified variable`

Here is an example of using Matlab to look at a wind stress dataset (tau.nc):

```
>> filename='tau.nc';
>> info=ncinfo(filename)
```

`info` contains structures that contain a whole bunch of information about the variables. It's confusing to unravel, so start with:

```
>> {info.Variables.Name}

ans =

    'tau'    'longitude'    'latitude'    'time'
```

which prints out the name of all variables in the file. You can then use `ncdisp` to find out more about these variables:

```
>> ncdisp(filename,'tau')

Source:
           /Users/kdrushka/class/tau.nc
Format:
           classic
Dimensions:
           longitude = 350
           latitude  = 60
```

```
            time      = 411   (UNLIMITED)
Variables:
    tau
            Size:       350x60x411
            Dimensions: longitude,latitude,time
            Datatype:   single
            Attributes:
                        units         = 'N/m^2'
                        missing_value = 1e+20
                        long_name     = 'wind stress magnitude'
                        short_name    = 'tau'
                        axis          = 'TYX'
```

```
>> taudata=ncread(filename,'tau'); % don't forget the semicolon!
This is a big variable.
>> lon=ncread(filename,'longitude');
>> lat=ncread(filename,'latitude');
>> time=ncread(filename,'time');
```

Let's also get the information about time:

```
>> ncdisp(filename,'time')

Source:
            /Users/kdrushka/class/tau.nc
Format:
            classic
Dimensions:
            time = 411   (UNLIMITED)
Variables:
    time
            Size:       411x1
            Dimensions: time
            Datatype:   double
            Attributes:
                        units       = 'days since 1950-01-01
00:00:00'
                        long_name   = 'Time axis'
                        time_origin = ' 1950-JAN-01 00:00:00'
```

The units of time are 'days since Jan 1, 1950'. Let's convert to the serial date number, i.e. the Matlab date format (days since Jan 1, year 0) – do this by adding an offset equal to the serial date number of Jan 1, 1950:

```
>> offset=datenum(1950,1,1)
>> time=time+offset;
```

```
>> size(taudata)
>> size(lon)
>> size(lat)
>> size(time)
```

`taudata` is 350 x 60 x 411, i.e. the dimensions of tau represent (lon,lat,time).

Note that the "missing value" is 1e20 (10^{20}). It's a good idea to set these to `NaN`:

```
>> missi=find(taudata > 1e19);
>> taudata(missi)=nan;
```

Matlab can only plot scalars, vectors, or 2-d data; `taudata` is 3-d. You must use indexing to select which parts of `taudata` to plot:

Plot data at the lon index 150 and the lat index 45, for all times:

```
>> plotstuff=taudata(150,45,:);
>> size(plotstuff)

ans =

     1     1   411
```

Note that Matlab preserves the 3-d structure from the original data (and thus will refuse to plot this). Use "squeeze" to get rid of the "singleton" dimensions:

```
>> plotstuff=squeeze(plotstuff);
>> size(plotstuff)


>> figure(1),clf
>> plot(time,plotstuff)
>> datetick('x')
```