Matlab bootcamp – Class 5

Kyla Drushka

Side-notes: scripts versus functions

Matlab program files (.m files) can be ***scripts***, which simply execute a series of Matlab statements (this is what we have done so far); or, they can be ***functions***, which accept input arguments and return output arguments. Both scripts and functions are Matlab code, and both are stored in text files with .m extensions. Note that all of the Matlab built-in functions (e.g. `mean`, `sin`, `plot`) are functions. It can be helpful to look at the code to get ideas about how to write your own functions: e.g. type **`open mean`** to see how Matlab computes the mean.

The first line of function has to be in this form:

```
function [output arguments] = FunctionName(input arguments)
```

There can be as many output and input arguments as you like (even none); their names are separated by commas.

Functions are useful when there is a computation or set of commands you want to do frequently (such as load a certain type of data file, or plot with a certain set of parameters): instead of having to include those commands in your code every time you want to do it, you write it in a function and "call" that function (i.e. use the function's name with the appropriate inputs and outputs).

Here is an example: a simple function to calculate the area of a cylinder, given radius and height as inputs. The area is returned as an output.

```
function A = cylinderarea(radius,height)

A=pi*radius.^2.*height;
```

Copy the text and save it into a file called cylinderarea.m (the function name is the name of the .m-file). Then, you can call the function by typing the function name and the input arguments. E.g. to compute the area of a cylinder with radius 3 and height 4:

```
>> cylinderarea(3,4)
```

This will call `cylinderarea`, assign the value 3 to the first argument (`radius`) and 4 to the second argument (`height`), and then run the rest of the code. The output (`A`) must be computed within the function.

You can use anything you want for the inputs: E.g.

```
>> cylinderarea(1,0)
```

or

```
>> R=2;
>> H=99;
>> cylinderarea(R,H)
```

The variables within a given script *only exist within the script* – this means that you either have to include them as inputs or create them within the script. And *only* the variables returned as outputs get saved in your workspace.

Other side-note: axis ij versus axis xy

`axis ij` specifies "matrix" axis mode: the coordinate system origin is in the upper-left corner (as for a Matrix);
`axis xy` specifies "Cartesian" axes mode: the coordinate system origin is at the lower left corner.

2-D plotting

So far we have just plotted lines. Here are some tools for 2-d plotting:

`imagesc(C)` or **`imagesc(x,y,C)`** – a flat image where each element of **`C`** specifies the color in one rectangular gridbox, and all gridboxes have the same size. with the color limits scaled such that the first color corresponds to the minimum value and the last color to the maximum value. The x/y ticks line up with the center of each box. By default, the y-axis is inverted (so maps appear upside-down and need to be flipped using the command **`axis xy`**). Pros: simple. Cons: NaN data are set to the lowest color (by default, blue).

```
>> load('sst_data.mat')
>> imagesc(lon,lat,sst);
>> axis xy
>> colorbar
```

`pcolor(C)` or **`pcolor(x,y,C)`** – a "checkerboard" plot where each element of **`C`** specifies the color in one gridbox, but the gridboxes can be different sizes and can be "shaded" differently using the command **`shading`**. The last row and the last column are cut off, and the x/y ticks line up with the leading edge of each box. Pros: pretty plots, different shading options, fast, gridboxes are stretched/interpolated, NaN values are white by default; Cons: bigger file

sizes, sometimes you don't want your data to be stretched/interpolated, it's confusing when the last row and column are removed.

```
>> figure(2),clf
>> pcolor(lon,lat,sst);
>> shading flat
```

Different options for **shading**:

```
>> figure(3)
>> C=rand(6,5);
>> subplot(2,2,1); pcolor(C);
>> subplot(2,2,2); pcolor(C); shading interp;
>> subplot(2,2,3); pcolor(C); shading flat;
```

contour(x,y,C) – contours the data in matrix **C** using 10 contour values. You can add an extra argument specifying the number of contour values , or the actual values to contour: **contour(x,y,N)** or **contour(x,y,V)** , where **N** is a scalar (e.g. **N** =100 will give 100 contour values) and **V** is a vector (e.g. [1 2 3 4]). Pros: interpolates, which can be desirable; also, you can return the values of the contour lines; good for plotting coastlines. Cons: can be very slow, looks awful if there are lots of NaNs.

```
>> figure(4),clf
>> contour(lon,lat,sst); xlabel('lon');ylabel('lat')
```

contourf(x,y,C) – is the same as **contour**, except the spaces between contour lines are filled with color. Same pros/cons as contour.

mesh(x,y,z,C) – produces a 3-D mesh surface such that the color represents the value of the matrix **C**, and the height represents the value of the matrix **z.C** and **M** must have the same dimensions. (If **C** is not specified, color is proportional to height). Use **view** to change the viewing angle. E.g.

```
>> figure(5),clf
>> mesh(lon,lat,sst); xlabel('lon');ylabel('lat')
```

Height corresponds to color.

Explore using a fixed height versus a fixed color (e.g. use a matrix of all ones for either the **z** or **C.**

```
>> all_ones=ones(size(sst));
>> figure(6),clf
>> mesh(lon,lat,all_ones,sst); xlabel('lon');ylabel('lat')


>> figure(7),clf
>> mesh(lon,lat,sst,all_ones); xlabel('lon');ylabel('lat')
```

surf(x,y,z,C) – is basically like **mesh**, but it uses the shading command also used with **pcolor**.

Pros for surf and mesh: allow you to visualize another dimension. Cons: can be hard to read the figures.

Note!

For most of these functions, Matlab returns an error if the dimensions of `x`, `y`, and `C` are not consistent, and if you have non-uniform spacing in `x` and `y`, `C` will be stretched accordingly. However, with `image` and `imagesc`, Matlab does not care what you specify for `x` and `y`, and will NOT stretch the image if `x` and `y` are non-uniform – so it is easy to plot an image with incorrect `x` and `y`! I like using `imagesc` because it generates images with smaller file sizes, but always make my plot with `pcolor` first just to make sure my `x` and `y` are consistent with `C`, and that `imagesc` is doing what I think it's doing.

You can use **hold on** with image plots as well as line plots: this keeps the current plot from disappearing when you plot something else on the same figure. E.g.

```
>> clf
>> imagesc(lon,lat,sst);
>> axis xy
>> hold on
>> plot(165,0, 's','markersize',10, 'markerfacecolor','w',
'markeredgecolor','k');
```

Image properties:

You can control many properties of the image.

colorbar – adds a colorbar

caxis`([cmin cmax])` – sets and min and max values of the color scale

colormap – sets "colormap", i.e. the set of colors used in the plot. See the help file for examples.

General Figure properties

These properties are can be used for all plots (not just images)

```
xlim([minXaxis maxXaxis])
ylim([minYaxis maxYaxis])

xlabel('label for x axis')
ylabel('label for y axis')
title('some title')
legend('line 1','line 2')
```

Note, you can combine multiple strings for the title and axes, for example if you want to include a number :

```
>> k=2;
>> title(['data set #' num2str(k)])
```

```
axis ij or axis xy
```

grid on or **grid off**

Adding text and annotations

You can add text to a plot using the command **text**, which has the syntax **text(xposition,yposition,'your text string')** , where `xposition` and `yposition` are relative to the current plot axes. You can also supply additional arguments, using the syntax **'PropertyName','PropertyValue'**, to control how the text looks. E.g.

```
>> text(165,3,'check out this sweet datapoint', 'fontsize',20,
'color','w', 'fontname','times')
```

The command **annotation** allows you to add text boxes, lines/arrows, and rectangles/ellipses. Note that **annotation** coordinates are specified relative to the *figure*, not the *plot*.

Get and set

It is possible to control nearly all aspects of your figures and plots using the command **set**. The syntax of **set** is:

set(H,'PropertyName1','PropertyValue1,'PropertyName2','PropertyValue2',...) where **H** is the "*handle*" of the "*object*" you want to modify. You can specify as many property name/value pairs as you like. *Objects* include figures, axes, and all the stuff in the plot (e.g. lines, contours, colorbars, titles, etc). Each object is assigned a unique number, which is called the "*handle*". The easiest way to figure out the handle of figures, axes, and objects is to

use one of the following commands (immediately after you have created the figure, axis or object):
`gcf` – returns the handle of the current *figure*
`gca` – returns the handle of the current *axis*
`gco` – returns the handle of the current *object*

E.g.

```
>> figure(1),clf
>> figH=gcf;
>> subplot(2,2,1);
>> axH=gca;
>> imagesc(lon,lat,sst);
>> colorbar;
>> cbH=gco;
>> subplot(2,2,2);
>> axH2=gca;
>> plot(lon,sst(10,:),'.');
```

Now, we have created variables corresponding to handles for the figure (`figH`), the two subplot axes (`axH` and `axH2`), and the colorbar object (`cbH`). Once these handles are stored as variables, any time we want to modify these properties we can just refer back to the handles using **`set`**. Eg.

```
>> set(axH,'xtick',[0 90 180 270 360],
'xticklabel',{'0E','90E','180E','270E'}, 'clim',[0 30])
>> set(cbH,'ytick',[0 5 10 12 14 16 18 20 25 30]);
```

Note that some plotting functions return the object handle as an output. E.g. when you add a colorbar using `colorbar`, if you supply an output, the colorbar object's handle is stored in that variable:

```
>> cb=colorbar;
```

`cb` is the object handle for the colorbar.

All figures, axes, and objects have a set of properties that you are able to modify with **`set`**, all of which have some default value as soon as the figure, axis or object is created. How do you figure out what these properties are, and what the current value of each property is? Use the command **`get`**:

```
>> get(figH)
```

Or, if you just want to "get" information about the current figure (whatever it happens to be),

```
>> get(gcf)
```

Similarly, to get information about the properties of the current axis:

```
>> get(gca)
```

The easier method: using the Figure Edit GUI

You can do all of this more easily (but less reproducibly) using the GUI: once you have plotted your figure, go to the Figure window and check out the different menu items, including:

Edit ➔ Figure Properties	to change the Figure name, colormap, background color
Edit ➔ Axes Properties	to change the title, grid-lines, axis limits and ticks
Edit ➔ Colormap	to play with the color map
Insert ➔ ...	to insert arrows, text, axis labels, etc

Important: once you have manipulated your figure to your liking using the GUI, you can then export the Matlab code that was used to generate the figure, using the menu item: File ➔ Generate Code

Mapping: the m_map package

Rich Pawlowicz at the University of British Columbia developed an awesome set of tools for creating maps. You have to download and install the package, and then you can access all the code. Link here: http://www.eos.ubc.ca/~rich/map.html

Make sure you have the folder while you stored m_map in your Matlab "path" (the path is the list of directories where Matlab looks for .m and .mat files)

```
>> path(path,'/Users/kdrushka/matlab/toolbox/m_map')
```

A quick demo – see the documentation for many more options. The basic principle is: 1) use **`m_proj`** to set up the map projection (e.g. Mercator, Lambert, Azimuthal, Sinusoidal) and the axis limits; 2) draw a coast line using **`m_coast`** ; 3) plot mapped data using **`m_plot`**, **`m_line`**, **`m_contour`**, **`m_pcolor`**, etc. 4) use **`m_grid`** to draw the grid for the given projection. E.g.

```
>> figure(1),clf
>> m_proj('ortho','lat',32.72','long',-117.16');
>> m_coast('patch','k');
>> m_grid('linest','-','xticklabels',[],'yticklabels',[]);
```

Plot our SST data:

```
>> figure(2),clf
>> m_proj('hammer-aitoff','clongitude',-150);
>> m_pcolor(lon-360,lat,sst);shading flat;
>> m_coast
>> m_grid
```