5.3 *Further (spatial / (x,y,z)) plotting*

As you have seen earlier – the simplest possible way of taking a matrix of data values and plotting them spatially, as a function of (*x*,*y*) location, is the function `image`. In effect, this is treating your data as if it were an image – the data values being the 'color' of each pixel and the location in the matrix defining where in the image (row, column) the pixel is. The problem with this is that information regrading what is on the *x* and *y* axes is lost, be this distance, lat/lon, or some set of observed/experimental variables, or whatever. instead, the points are evenly spaced on both axes. Moreover, the raw values are plotted and there is no possibility of interpolation/contouring or smoothing. One could regard scatter plotting as an improvement over this and a sort of *x*,*y*,*z* plotting, in as much as a 3rd dimension (*z* data value) can be represented through color and/or symbol shape and at time this can be quite effective. However, again, no interpolation/contouring or smoothing is possible with `scatter`.

For plotting true (*x*,*y*,*z*)/'3D' plots (i.e. data values in 2 spatial dimension), MATLAB provides a wide variety of more formal ways of plotting data spatially, including even the possibility of adding a 4th dimension representing the data value (*x*,*y*,*z*,*zz*) (see Box).

For a feel of what you should be able to learn to achieve using **MATLAB** – go to the following webpage. In this data repository you can do things like re-plot with different longitude, latitude, and temperature ranges. Overlay the coastlines, and other useful things like that. You can also click through the different months of the year to get a feel for how the surface temperatures on Earth change with the seasons. Note that the graphic produced from this particular website is not particularly great, and you can all do better than this using **MATLAB** already. Presumably there are some lazy PhD students out there lacking the skills that you are (hopefully) learning. Perhaps they should take GEO111 (or maybe you are ...)?

x,y,z PLOTTING

MATLAB calls plots of a (*z*) value as a function of both *x* and *y*, '3D'. Strictly, one could look at some of these functions as 2D, as scatter can plot a 3rd data (*z*) value as different colors/shapes/sizes as a function of both *x* and *y* ... Anyway, the most commonly used/useful and fortunately simple, functions which create a 2D (*x*, *y*) plot but with contours in the value of (*z*), are:

1. `contour` – Plots a figure with the data contoured, with a range and increment between contours that is fully specifiable, color-coded or not, and labelled or not. Options are also provided for specifying how the contouring is done (and the data interpolated).
2. `contourf` – Similar to `contour`, except in between the (now simple black, by default) contours, a fill color is plotted and scaled to the data value.

For a genuine 3D plot, with surface height determined by the data in the 3rd dimension of the array, colors and/or contours in the data in the 4th array dimension, suitable functions include:
`surf`, `surfc`, `mesh`
(but are not considered further here).

As an example, load in the global topographic data file (**etopo1deg.dat**) from the course webpage. This is the height of the (solid) surface of the Earth relative to mean sealevel in meters, with the continents having a positive value and the ocean floor, negative. The data is conveniently on a 1° (longitude and latitude) grid. You could view the resulting elements of the 2D array in the Variable window if you like ... but at 360×180 in size, there may not be much of use you can glean by visually inspecting the matrix[7].

Try throwing the array into the `image` function see what happens (hopefully something like Figure 5.1). It it had happened to come out

`imagesc` For a data array (matrix) `A`,

```
imagsc(A)
```

displays the data array as if a bitmap, but unlike `image` (see earlier), *"uses the full range of colors in the colormap*".

[7] More useful then are the summary details in the **Workspace window**, such as the apparent absence of NaNs and that the **Min** and **Max** Earth surface heights seem plausible.

displayed upsidedown[8], then you'd need to flip the matrix upside-down using the command:

```
etopo1deg=flipud(etopo1deg);
```

and if the Earth instead appeared on its side[9], you need to swap the rows and columns (x for y axis):

```
etopo1deg=etopo1deg';
```

It is not unusual for a first plotting attempt of spatial data to be incorrectly orientated and a little trial-and-error to get it straight is perfectly acceptable!

This is not exactly the prettiest of images. You can distinguish ocean (blue) from land (mostly brown, but other color pixels in places). Fortunately, MALTAB provides a variant of this plotting function, `imagesc`, that calculates the color scale to exactly span the min/max values in the data. Try it (and get something like Figure 5.2 hopefully).

The function `imagesc` also enables the range of data values the colo9r range corresponds to, to be set. Refer to `help` on this function and see if you can plot just the above-sealevel, i.e. land surface heights, spanning zero (sealevel) to the maximum height[10].

Which sort of in a round-about sort of way also brings us to how to set the color scale, which can be changed using the `colormap` command (see Box). Try out some different *colormaps* and re-plot the global topography data.What scales work well and what do not? Which scales help pick out details of e.g. ocean floor depth variation and which help pick out simple land-sea contrasts. Think about what one might want to highlight about global topography and what color scale might be best for this purpose?

Sticking with global Earth surface topography, how else can we display the spatial data? For instance we might want to interpolate it, contour it, or simple get the longitude and latitude exes correct. Note that only by luck, because this particular dataset is 1 degree by 1 degree, the default axis scale in MATLAB when using image is approximately correct, although note that 'latitude' has been ordered in reverse and it goes from 1 to 180 rather than -90 to 90 ... We'll explicitly address this shortly.

To start with, you can simply use the `contour` function (see Box), passing only the matrix (of global topography values). Try this. Now you might want to think about flipping the matrix up-down, and/or left-right, as your plot should have come out looking like Figure 5.3.

Once you have fixed the orientation of the topography map, you might play about with the color scale (`colormap`) as before. You

[8] It doesn't in this particular case.

[9] Actually, in this example, it is OK in this respect too. Boring!

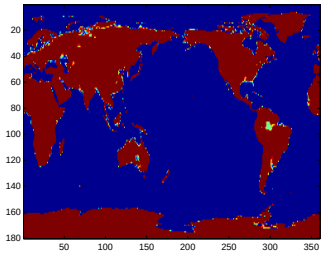

Figure 5.1: Very basic imaging (`image`) of an array (2D) of data – here, global bathymetry.

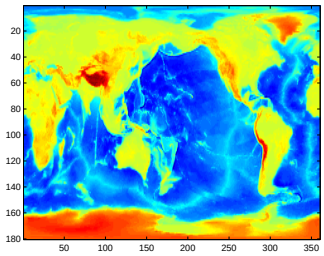

Figure 5.2: Slightly improved very basic imaging (`imagesc`) of bathymetry data.

[10] Don't forget the function `max`.

`colormap` **MATLAB** has a number of 'colormaps' built in – color scale that determine the colors that correspond to the data. The command to change the *colormap* from the default is:

```
» colormap NAME
```

where `NAME` is the name of the *colormap*. You can find a list of possible *colormaps* in **help** on `colormap` (in a table towards the bottom). But a brief summary is:

- `parula` – the current **MATLAB** default – chosen to provide a wide range of color and color intensity.
- `jet` – the old **MATLAB** default, but one which uses red and green in the same color, which should be avoided (why?).
- `hot`, `cool` – relatively simple color transitions but useful – hot is something like you'll see in publication figures.
- `pink` – another simple and at times useful transition and from dark (almost black) to white.

To return to the default *colormap*:

```
» colormap default
```

might also try the companion to `contour` – `contourf`. Re-orientating the matrix, switching to a different *colormap*, and plotting using `contourf`, might give you something like Figure 5.4.

OK, so a next refinement in plotting esp. maps and contour plots, is firstly to specify the range of the color scale, as we may not want the min-to-max range chosen by default by **MATLAB**, and the number of contours (e.g. in the topography example, they are pretty far apart and it is difficult to make out much detail). Both of these factors can be addressed simultaneously, by giving MATLAB a vector containing the value at which you want the contours drawn[11].

Taking the global topography data – lets say you were interested only in low lying and shallow bathymetry, and wanted 20 contours intervals. Assuming a range in topographic height (relative to sealevel) of -1000 m to +1000 m, you should be able to deduce how to create the vector(?)[12]

Do this and check e.g. by opening up the vector in the **Variables window**. You should see the numbers from -1000 to 1000 in intervals of 100. Why, for instance, can you not simply write:

```
» v = [-1000:1000];
```

??? (Or rather: why might this not be a good idea ... ?)

Having created a specific vector of contours to plot, try it out. OK – so this is a little weird and maybe not so useful, but you get the point hopefully. So try plotting the following:

1. Just above sealevel topography, up to 10,000 m, in increments of 100 m.
2. Just the sealevel (coastline) contour ... trickier – create a vector with a value at zero, and a value either side – one very high and one very low. Use `contour` rather than `contourf`, although the latter produces a lovely land-sea mask!
3. Convert the data matrix of value in units of m, to ft, and plot the ocean floor (values equal to or below sealevel) in intervals of 1000 ft.
4. Finally – try some different color scales for the above. Think about which color scales best help illustrate the data, and whether `contour` or `contourf` is clearer. Also: how many contour intervals is 'best'? You key is to make features clear, within the plot becoming cluttered or overly detailed.

The final refinement in contour plotting we'll look at is adding labels to the contours. The command to do this is `clabel` (for 'contour label') (see Box). Now, before anything, there is a slightly complication. `clabel` needs to know details of the contours and graphics object with which to do anything with. For the purposes of this course,

[11] By default: MATLAB determines the minimum and maximum data values, and draws 10 equally spaced contours between these limits.

[12] If not, it is:

```
» v = [-1000:100:1000];
```

`contour` There are various uses of contour. The simplest is:

```
contour(Z)
```

where `Z` is a matrix. Thsi ends up similar to image except with the data contoured rather than plotted as pixels (the 'simularity' here is that the x and y axis values simple are the number of the rows and columns of the data).

You can specify the values at which the contours are drawn, by passing a vector (v) of these values, e.g.

```
contour(X,v)
```

More involved and practical, is:

```
contour(X,Y,Z)
```

where `X`, `Y`, and `Z`, are all matrices of the *same* size (there is important). `X` and `Y` contain the x and y coordinate locations of y data values (contained in matrix `Z`). In the example of a map – `X` and `Y` contain the longitude and latitude values of the data values in `Z`.

Similarly, you can add a vector `v` containing the contours to be drawn, by:

```
contour(X,Y,Z,v)
```

you don't have to worry about the details of this, but simply need to know the following:

1. When you call `contour` (or `contourf`), 2 parameters are returned, which so far you have not cared about or even noticed. We now need them. SO when you call either potting function, using the syntax:

   ```
   [C,h] = contour( ...  )
   ```

   which saves a matrix of data to `C`, and a ID (technically: graphics object 'handle')to `h`.
2. When you call `clabel`, pass these parameters back in, e.g.

   ```
   clabel(C,h)
   ```

(in its most basic usage).

If you do this, in an earlier example of plotting just the zero height contour, and now using the most basic default usage of clabel (as above), you get, for good or for bad, Figure 5.5.

In the default usage of `clabel`, you'll get a label added on every contour that you plot. This ... can get kinda messy if you have lots and lots of contours plotted. You may well not need every single contour labelled, particularly if you also provide a color scale (see below). So you can also pass in a vector to tell MATLAB which contours to label. For example, if you have a contour interval vector:

```
v = [-1000:100:1000];
```

maybe you onyl want labels every 500m, so you'd use a vector:

```
w = [-1000:500:1000];
```

to specific the labelling intervals. The complete set of commands becomes:

```
» v = [-1000:100:1000];
» w = [-1000:500:1000];
» [C,h] = contour(etopo1deg,v);
» clabel(C,h,w);
```

Finally – missing from our color-coded plots so far, is a color scale to relate values to colors (although labelling the contours works as an OK substitute). The MATLAB command is simple:

```
» colorbar
```

(and see Box for further usage). Try adding a *colorbar*, and in different places in the plot. Refer to the Box to try and add a caption to it ...

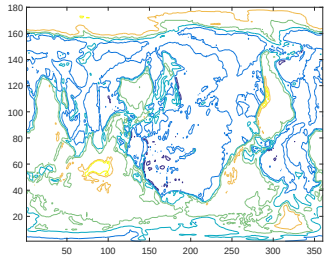

Figure 5.3: Example result of basic usage of the `contour` function.

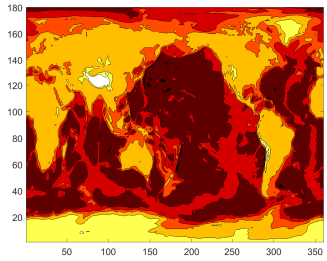

Figure 5.4: Example usage of `contourf`, with the `hot` *colormap* (giving dark-/brown colors as deep ocean, and light/white as high altitude).

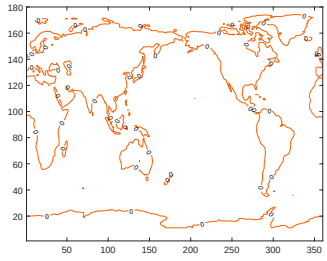

Figure 5.5: Example usage of `contour`, contouring only the zero height isoline, and providing a label.

`clabel`

```
» clabel(C,h)
```

labels every contour plotted from

```
[C,h] = contour( ...  );
```

(or from `contourf`).

By prescribing and passing a vector `v` of contour intervals, you can label fewer/specific intervals rather than all of them (the default), e.g.

```
» clabel(C,h,v)
```

In this next Example, we'll address the issue with missing/incorrect lon/lat axis labels on the plots.

Each data point in the `etopo1deg` matrix should have one longitude value (x-axis) and one latitude (y-axis) value associated with it. It should hopefully be intuitive to you now ... that what we need is a pair of matrices, of exactly the same size as the `etopo1deg` data matrix – one holding longitude values and one latitude values. There are various ways of creating the required matrices 'by hand' (or involving writing a program including a *loop*). All of them are tedious. There is a **MATLAB** function to help. But it is not entirely intuitive[13] ... `meshgrid`.

Spend a few minutes reading about it in `help`. In particular, look at the examples given to help you translate the **MATLAB**-speak gobbledegook of the function **Description**. You should be able to glean from all this that this function allows us to create two $a \times b$ arrays; one with the columns all having the same values, and one with the rows all having the same values (exactly what we need for defining the (lon,lat) of all the global data points). If not, and probably not – see Box. And then lets do a simple example (adapted from **help**):

```
» [X,Y] = meshgrid(1:3,10:14)
X =
     1  2  3
     1  2  3
     1  2  3
     1  2  3
     1  2  3
Y =
     10  10  10
     11  11  11
     12  12  12
     13  13  13
     14  14  14
```

Here, we are taking 2 vectors – `[1:3]` and `[10:14]`, and asking **MATLAB** (very nicely) to create 2 matrixes, one in which `[1:3]` is replicated down, until it has the same number of rows as the length of `[10:14]`, and one in which `[10:14]` is replicated across until it has the same number of columns as the length of `[1:3]`. (Try it.)

It'll become apparent *why* bother shortly. Honest.

In our Example – start my noting that the topography data is on a regular 1 degree grid starting at 0° longitude. Latitude starts (at the bottom) at -90° and goes up to +90°). We need a matrix containing all the longitude values from 0° to 359° and latitude from -90° to 89°

`colorbar`

This almost could not be simpler:

```
» colorbar
```

plots the color scale! By default, is places it to the RH sice of the plot. If you wish for it to appear anywhere else, use the modified syntax:

```
» colorbar(PLACEMENT)
```

where PLACEMENT is one of: `'northoutside'`, `'southoutside'`, `'eastoutside'`, `'westoutside'`. Note that these are strings and so need to be in quotation marks. (More options are summarized in a table in **help**.)

Finally, you can also add a label to the *colorbar*, but only if you get hold of its ID ('graphics handle') when you call `colorbar`, e.g.

```
» h = colorbar
```

will save the graphics handle in variable `h`, which you can then muck about with via:

```
c.Label.String = 'The
units of my lovely
colorbar';
```

(Don't fight this – use this syntax to set a label for the *colorbar* – don't worry about what it means. **MATLAB** keeps rather annoyingly changing the way it does this anyway :()

[13] **DON'T PANIC!**

`meshgrid`

The unholy syntax is:

```
[X,Y] = meshgrid(xv,yv)
```

Pause, and take a deep breath. On the left – the results of `meshgrid` are being returned to 2 matrixes, `X` and `Y`. These are going to be our matrixes of the longitude and latitude values (in the particular example in the text). So far so good(?)

On the right, passed into the function `meshgrid`, are two vectors – `xv` and `yv`. Pause again.

What **MATLAB** is going to do, is to take the (row) vector `xv`, and it is going to replicate it down so that there are as many rows as in the vector `yv`. This becomes the returned output matrix `X`. **MATLAB** then takes the column vector `yv`, and replicates it across so that there are as many columns as in the vector `xv`. This becomes the returned output matrix `Y`.

.[14] These matrices need to be the same size as the data matrix.

Maybe just 'do' it and then understand what has happebned after. Create the longitude and latitude grids by:

```
» [lon lat] = meshgrid([0:359],[-90:89]);
```

View (in the **Variables window**) the `lon` matrix first. Scan through it. Hopefully ... you'll note that it is 360 columns across, and in each column has the same value – the longitude. The matrix is 180 rows 'high', so that there is a longitude value for each latitude. Similarly, view `lat`. This also should make a little sense if you pause and think about it, with the one exception that the South Pole latitude is at the 'top' of the matrix – don't worry about this for now ...

The only way to fuly make sense of things now, is to use it. Remember that use of `contour` (and `contourf`) can take matrices of x and y (here: longitude and latitude) values that correspond to the data entries in the data matrix (`etopo1deg`). Re-load the topography data in case you have flipped it about in all sotes of odd ways, and then do:

```
» [lon lat] = meshgrid([0:359],[-90:89]);
» contour(lon,lat,etopo1deg);
```

Almost! Note that the x and y axis labelling is 'correct' and particaulrly the y-axis, where latitude gos from -90 to 90 (although by default **MATLAB** labels in intervals of 20 starting at -80 it seems). But it also turns out that we do need to flip the data op-side-down. We can actually do this in the same line as we plot:

```
» contour(lon,lat,flipud(etopo1deg));
```

Phew! (Figure Figure 5.6.)

The final complication is that the data points in the gridded dataset (matrix `etopo1deg`), technically correspond to the mid-points of a 1 degree grid, not the corners. So if we were going to try and be formally correct[15], our vectors that we'd pass into `meshgrid`, would be:

```
» xv = [0.5:359.5];
» yv = [-89.5:89.5];
```

OK – another Example on this. Previously, you downloaded and plotted monthly global distributions of surface air temperature. You plotted these simply using `pcolor` (or `image`) and the results were ... variable. Certainly not publication-quality graphics and missing appropriate longitude and latitude axes for the plots.

[14] There is a slight complication with this, which we'll get to shortly, but note that the data array is 360 elements (x-direction) by 180 elements (y-direction).

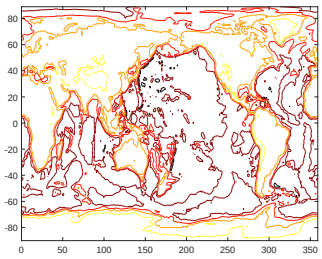

Figure 5.6: Usage of `contour` but with lon/lat values created by `meshgrid` function and passed in (and with the `hot` *colormap* (giving dark/brown colors as deep ocean, and light/white as high altitude).

[15] Don't worry about this for now – grids will be covered more in subsequence chapters surrounding numerical (environmental) models.

Make a copy of your original *script* (**m-file**) in which you created the animation, and give it a new name. Edit your program, and in place of `pcolor`, use `contour` or `contourf` (your choice!). Pass in just the data matrix (of monthly temperature) when calling the `contour(f)` function and don't yet worry about the lon/lat values. Get this working (i.e. debug it if not). You should end up with a contoured animation (rather than a bit-map animation).

The problem with the axis labelling should be much more apparent (than compared to the topography data, which was on a handy 1 degree grid already). So you need to make a matrix of longitude values, and one of latitude. using `meshgrid`. The grid is a little awkward:

1. The longitude grid runs from 0°E (column #1) with an increment of 1.875°; i.e., 0.000°E, 1.875°E, 3.750°E, ... up to 358.125°E (column #192).
2. Latitude runs from 88.54196°S (-88.54196°N) at row #1, to 88.54196°N (row #94) with an increment of about 1.904.

so I'll give you the answer up-front:

```
» lonv = [(1.875/2):1.875:360-(1.875/2)];
» latv = [-90+(1.904/2):1.904:90-1.904];
» [lon lat] = meshgrid(lonv,latv);
```

Now use the longitude and latitude values matrices, in conjunction with `contour(f)`, to plot the global temperature distributions 'properly'. Try plotting just one plot first, before looping through all 12 months.

At this point (before creating an animation), you might also explore some of the plotting refinements we saw earlier. For example, as per Figure 5.7. Firstly – get the units of the temperature data array into units of °C or °F rather than °K. Either: assign the `temp` array data to a new array and make the appropriate conversion from °K (all within the loop), or you can do this subtraction on the line that you actually plot the data (i.e., within the `contour/contourf` function), for example:

```
contourf(lon(:,:),lat(:,:),temp(:,:,month)-273.15);
```

would convert to °C as it plotted the data.

You can also get the plotting temperature limits and contouring consistent between months and with greater resolution by adding the following line (before the loop starts):

```
v=[-40:2:40];
```

and then to the `contour(...)` (or `contourf(...)`) function, add `,v` to the end of the list of passed parameters. This particular choice for the vector `v` tells MATLAB to do the contouring from -40 to 40 (°C), and

at a contour interval of 2 (°C).. Play around with the min and max limits of the range, and also with the contour interval to see what gives the clearest and least cluttered plot. For instance, maybe you don't want the low temperatures to go 'off' the scale (the white color in the filled contour plot).

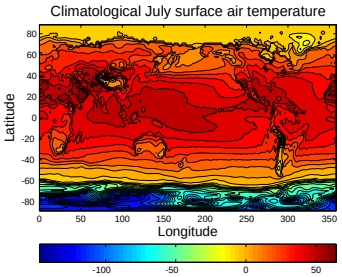

Figure 5.7: Example contour plot including `meshgrid`-generated lon/lat values. Result of `contourf(lon,lat,temp7,30)`, where the data file was `temp7.tsv`, with some embellishments.

5.3.1 *Plotting maps*

You can do some nice spatial plotting with this data using the **MATLAB Mapping Toolbox**. This should be available as part of the **MATLAB** installation in the Lab (and also if you have downloaded and installed an academic version on a personal laptop). Refer to the online documentation for the **Mapping Toolbox** to get you started. The key function appears to be `geoshow`. Try plotting the region encompassing the 'quake data, with a coastal outline (of land masses), and the 'quake data overlain. Explore different map projections. Remember to always ensure appropriate labelling of plots.