Guide to Klinger Graphics Scripts

Barry A. Klinger May 20, 2015
George Mason University AOES Dept., bklinger@gmu.edu

This document gives advice on

- How to make scientific graphics clearer.
- A collection of Matlab mscripts I have written to assist in producing improved figures with Matlab.

This guide is *not* an introduction to Matlab. Many of the ideas discussed here will translate to other programming languages too. This guide does not discuss all the mscripts in the collection; see a separate listing for these.

"Beautiful graphics" can have at least two meanings in scientific communication. One is the aesthetic values of the figures, how they are decorated, how the space around them might be designed to achieve a certain psychological effect, etc. As far as I can tell, these are all pretty irrelevant to the goal of the figure and can actually get in the way of good communication. The second is how well the figures explain something. Generally the purpose of such a figure is to answer some question. A plot may seem complete, but if the reader can't find key information because the axis labels are too small to decipher, or the figure is too crowded, or a key feature is not visible because of poor choice of contour values or colors, then the figure has not achieved its purpose. Here then are some techniques to help create beautiful graphics - in the second sense of the phrase.

Contents

1 Use of Colors to Represent Fields **2**

2 Contour Plots: contourfP.m, colorpal.m, maptoint.m **5**

3 Vectors and Surfaces **8**

4 Figure and Axis Control **9**

5 Map Projections **15**

1 Use of Colors to Represent Fields

Matlab coloring of surfaces with contours (**contourf.m**) or pixel-coloring (**fill.m**) maps numerical values to colors via a **colormap**. The default colormap is **jet**, which has a color sequence similar to a rainbow. The Matlab mapping to the rainbow palette has a number of problems which are solved with my routine **contourfP.m** and related mscripts.

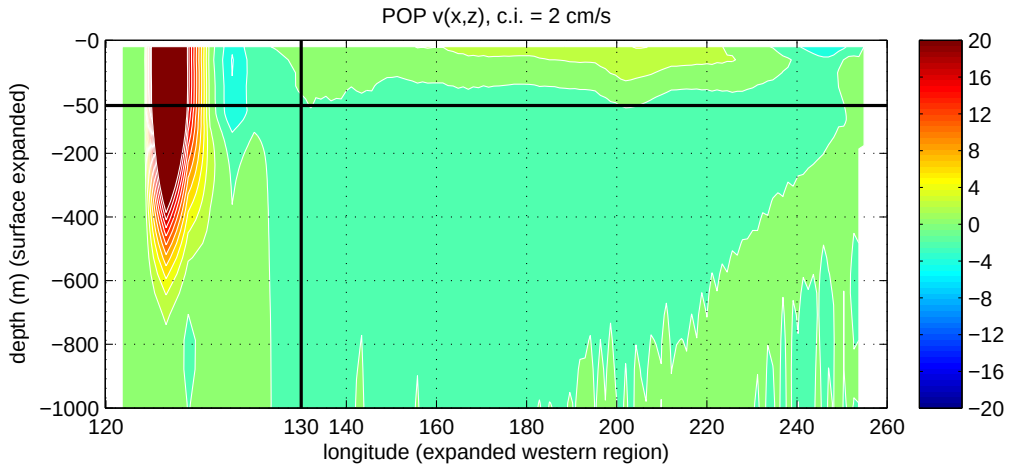

Figure 1: **Problematic Matlab output:** Annual average meridional velocity from the POP ocean general circulation model in the North Pacific, uniform contour interval and **jet** colormap. From **fPdemo1.m**.

Using rainbow colors for representing numerical data is problematic for several reasons (Light and Bartlein, 2004, *Eos*, 85, pp. 385, 391)[1] . The rainbow palette relies on transitions between different colors, including blue and green, but a significant minority of men have trouble distinguishing blue and green due to colorblindness. More fundamentally, numerical fields such as temperature or speed are usually approximately continuous, but humans perceive colors as falling into a small number of distinct groups (blue, green, yellow, etc.). Therefore there is a natural tendency to infer special signifance to transitions between different groups (for instance a border between yellow and green). This can be misleading in many graphs.

The rainbow palette is also ill-suited for displaying data for which we want to clearly show the difference between positive and negative values. Fig. 1 illustrates this problem; even with the colorbar next to the figure it is hard to tell exactly which shade of green is positive and

[1]Thanks to Jennifer Adams at COLA for pointing out sources on this subject and for subsequent discussion.

which is negative. We could attach labels to the contours but if the colors are not helping the viewer extract information from the figure, why bother having the colors?

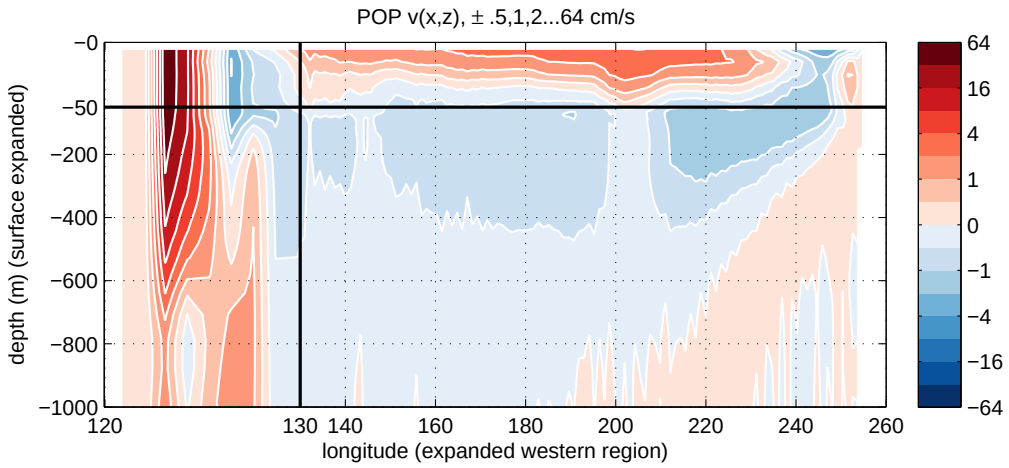

Figure 2: Annual average meridional velocity from the POP ocean general circulation model in the North Pacific, exponentially growing contour intervals and blue/red color scheme. From **fPdemo1.m**.

A related problem in using contours or colors to represent an array of numbers is that sometimes very different contour intervals are needed for different ranges of numbers. In Fig. 1, most of the section has small v, but a small region has a large value. Both regimes contribute comparable amounts to the total transport. A uniform contour interval will be too big to resolve structures in the small-v regions or so small that the region of large values will be cluttered or saturated. Fig. 1 actually has both problems! Even if variable contour intervals are used, the Matlab colormap will make the color difference across small contour intervals much smaller than the color difference across large intervals. Often in this case the colors between many of the contour levels become indistinguishable.

Illustrating a better alternative to the rainbow palette, Fig. 2 uses different families of shades for positive and negative values, unevenly spaced contours, and comparable steps in shading across all contours. This set of choices make positive and negative parts of the field obvious, reveals previously-hidden structures such as the relatively strong negative values in the east, and allows the viewer to see the magnitude of the field at all locations. The contour values increase exponentially in magnitude, taking values of $0, \pm.5, \pm1, \pm2...\pm64$. The exponential sequence allows us to vary the contour interval over a wide range but in a regular and understandable way. A similar way to show positive and negative values is to shade for the magnitude and color the contours for the sign, as in Fig. 3.

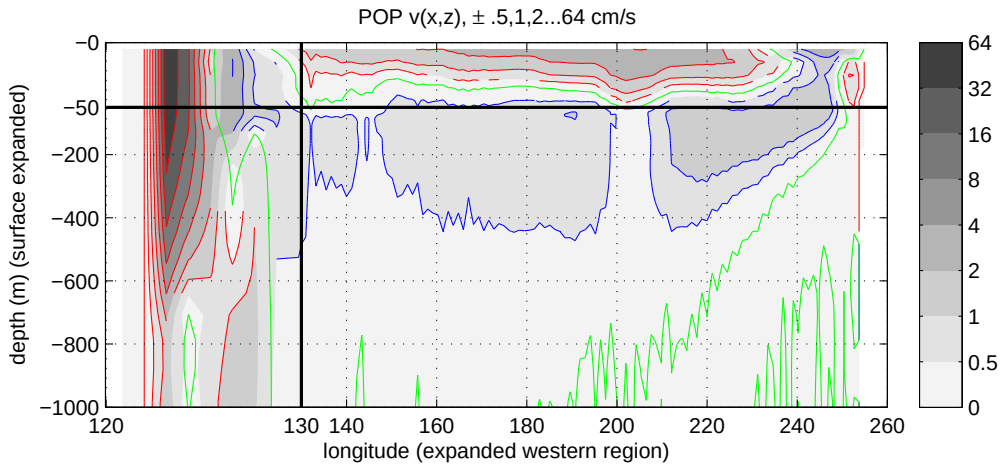

Figure 3: Annual average meridional velocity from the POP ocean general circulation model in the North Pacific, exponentially growing contour intervals, gray shades for magnitude, and and blue/red/green contours for positive/negative/zero contour values. From **fPdemo1.m**.

Besides differentiating positive and negative regions of a field, different colors can be used to emphasize features of the field. In Fig. 4, water from surface sources in the northern and southern hemispheres are colored in shades of orange and green, respectively. Because small salinity gradients are more significant in the deep water than near the surface, the contour interval is smaller between 34 psu and 35 psu which corresponds to deep water. One should use caution in using colors and uneven contour intervals in this way because it is easy to give a misleading impression by choosing colors or contours in a certain way. In Fig. 4 the different contour intervals are marked by using thicker lines for contours that are separated by the larger interval. Generally the figure caption or text should flag the use of non-uniform contours or sharp color transitions in order to avoid misleading the reader.

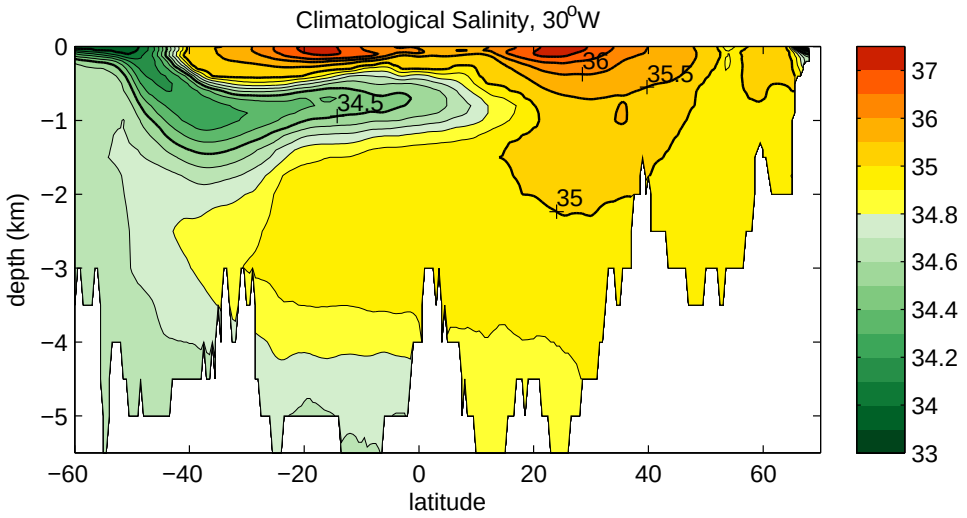

Figure 4: Ocean salinity (from World Ocean Atlas 2001), using two colors and two different contour intervals.

2 Contour Plots: contourfP.m, colorpal.m, maptoint.m

A powerful way to make filled contour plots with the features displayed in the previous section is to use the mscript **contourfP.m**. This script creates a colormap from a collection of pre-assigned palettes. It associates each contour interval in a sequence of contours with a color in the colormap. There is an option to use gray shades for magnitudes with different color contour lines for positive, negative, and zero values. There are also options to modify the linewidth and to modify the palette.

```
[cs,han]=contourfP(x,y,A,ci,Fcol,xBrk,Lcol,lthick,xlohi);
%   (x,y,A,ci) = coords, array, and contour values as in contourf.m.
%   Fcol = string of letters representing sequence of colors.
%   xBrk = for each color in Fcol, corresponding element in vector
%          xBrk tells which value of ci marks beginning of next color.
%   Lcol = line color, 'none', single letter, or 3 letters for +/-/0.
%   Lthick = line thickness (default = .5)
%   xlohi = array for fraction of palette used, each row of array
%          corresponds to color in Fcol and modifies range of
%          shades (ex: [.1 .8] goes from 10% to 80% through palette)
```

A useful script to generate an exponentially varying vector of contour values (including negative and zero values) is

```
ci=expci(cimin,N,base);
% raise base (default=2) to powers 0 to N-1 and multiply by +/- cimin
```

The key part of the script **fPdemo2.m** to make Fig. 2 is relatively simple:

```
ci=expci(.5,8);
contourfP(xnon,-znon,vv,ci,'bR',0,'w',1,[.1 1; .1 1]);
```

The text **bR** creates a colormap consisting of blue shades (b) followed by red shades (R), with lower case letters representing palettes that go from dark to light as data values increase, and upper case letters representing palettes that go from light to dark. The parameter **0** tells the script to switch colors from blue to red at ci=0. The **w** makes contour lines be white, and the **1** gives the linewidth of the contours. The colors are determined from standard palettes by **contourfP.m** calling **colorpal.m** (described below). In this example, the .1's make the lightest colors in the palette be a little darker than the lightest color in the base palette.

The main code to make Fig. 3 is

```
ciG=expci(.5,-8);   % negative N ==> exclude negative values
contourfP(xnon,-znon,vv,ciG,'A',0,'rbg',.5,[.1 .8]);
```

Here the **A** specifies a grAy palette, and the use of three letters for the contour colors (in this case **rbg** for **red-blue-green**) tells the mscript to use the stated colors for positive, negative, and zero contours, respectively. Note that for this plot, the contour lines are thinner and darkest shades are lighter than in Fig. 2.

The color palettes used in the **contourfP.m** examples above were created with a routine called **colorpal.m** which returns an $N \times 3$ array which can then be used to set a colormap:

```
colpal=colorpal(col,ncol,xlohi);
colormap(colpal);
```

The first input parameter **col**, is a string representing the colors to be used in the colormap. If the first letter of the string is **h** or **H**, calling the function will write help notes to the Matlab window and will display all the available palettes in a graphics window (Fig. 5, top row). The palettes are largely (but not entirely) based on those shown at the website colorbrewer2.org. The vector **ncol** gives the number of shades to be used for each color. If the length of **col** is greater than the length of **ncol**, then **colorpal** will plot the colors represented by the output vector **colpal** (Fig. 5, bottom two rows). Shades in each palette

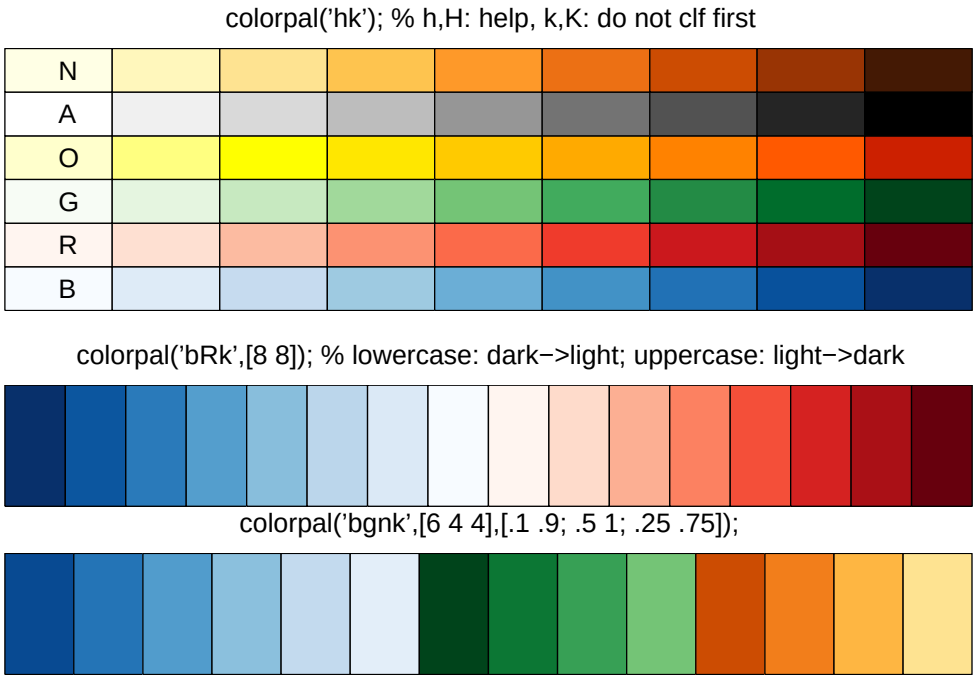

Figure 5: Top: available colormap palettes available through colorpal.m. Letter used to call each color are given at left in the panel. Middle: example of a blue-red colormap. Bottom: example of a blue-green-brown colormap with the range of shades altered by the last parameter in the function. From **colorpaldemo.m**

are interpolated from each family of 9 shades shown at top of Fig. 5. The range of shades can be compressed with input array **xlohi**; each row of xlohi has two numbers, each between 0 and 1, which represent the minimum and maximum fraction of the entire shade range will be used.

In **contourfP.m** the colors are assigned to integers 1,2,..., so in a **colorbar** the labels will default to these values. To give the correct labels (here selected as every other contour values), **colorbar** must be called with some updated parameters, for instance:

```
ibar=1:2:length(ci);
colorbar('ytick',ibar,'yticklabel',ci(ibar))
```

3 Vectors and Surfaces

Matlab provides a script called **quiver** for graphing two-dimensional vector fields. Such fields present a graphical challenge because there is a tension between showing each vector large enough to see it clearly and showing all the vectors in a wide region. Figure 6, upper left, shows a typical example. While this figure conveys the main idea—there is a strong current leaving the coast and flowing to the northeast—it's hard to learn more than that because the arrows in the strong currents are crowded and the arrows in the weak currents are nearly invisible. The upper right figure reduces the crowding by skipping every vector, but the low resolution makes it hard to tell where the edges of the strong current are. My routine **quivcheck** (lower left) omits data in a checkerboard pattern. I find this is often a good compromise. Especially for arrows that mostly point in the horizontal or vertical (in the figure) directions, the checkerboard pattern reduces crowding while preserving features. Of course, for very high resolution data it may be best to reduce the resolution (as in upper right panel) *and* use **quivcheck**. Finally, my colleague Jay McCreary taught me that letting arrow length be proportional to the square root of the vector magnitude (instead of it being proportional to the vector magnitude), as in the lower right panel, allows us to see both the strong and weak currents in the same plot. This can be done with my routine **quivsqr**, which is also called by one option of **quivcheck**.

For conveying quantitative information, 2D contour or shading plots are usually better than perspective 3D plots. Occasionally one may want to convey a more visually appealing impression with a perspective plot. Matlab routines such as **surf** (Figure 7, upper panel) and **mesh** (not shown) illustrate a 2D surface as if looking at it in 3D. They smoothly connect each value in the array. Sometimes one wants to emphasize that each array value represents a grid box with a constant value. This can be shown by transforming the data with my routine **plateau** and plotting the result with **mesh** (bottom panel) or **surf**.

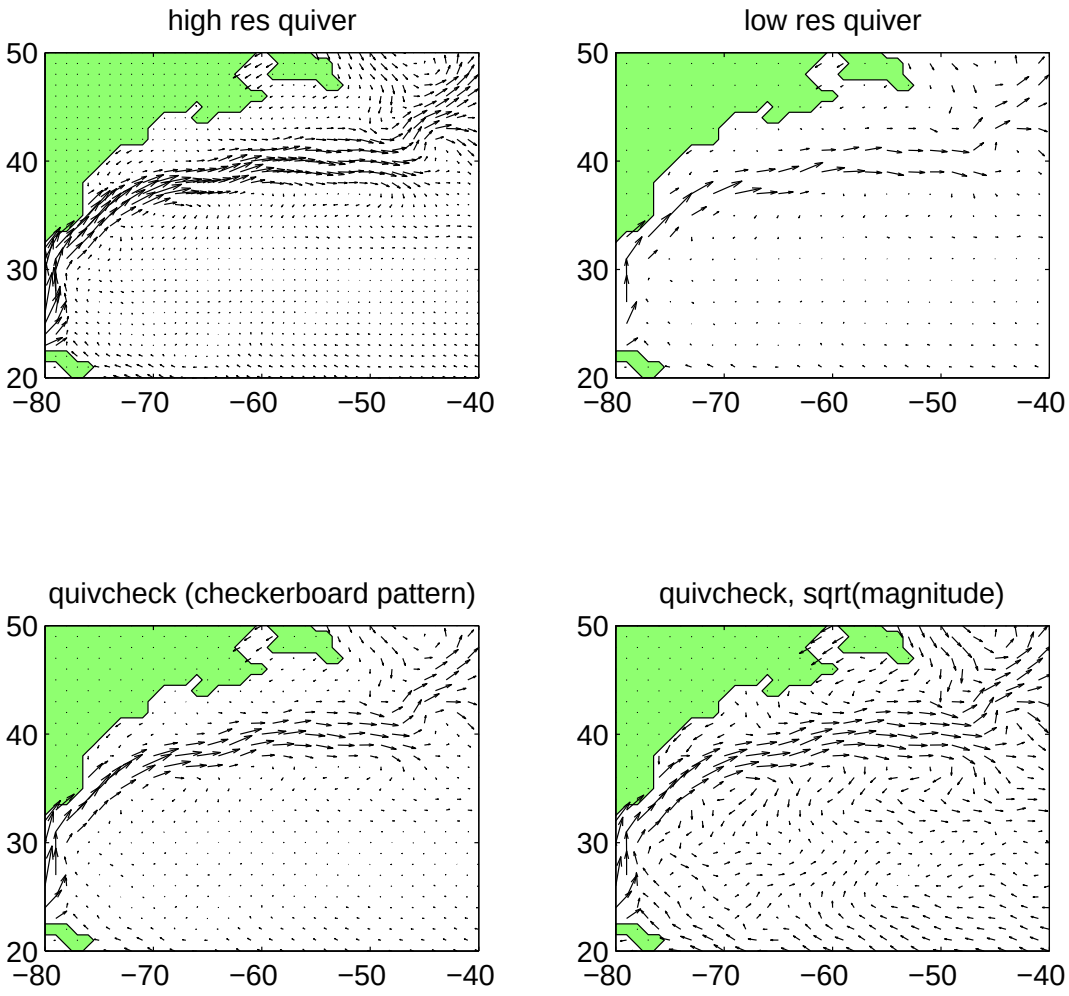

Figure 6: North Atlantic surface velocity climatology to illustrate various **quiver** options (**quivdemo**). Each point shows average of 3X3 set of gridpoints centered on the point. Data is from www.aoml.noaa.gov/phod/dac/drifter_climatology.html (Lumpkin and Garraffo, 2005: J. Atmos. Oceanic Techn.).

4 Figure and Axis Control

A number of my scripts to control figure and axis characteristics are very short but I use them so often that it is still useful to call them rather than to write out the few lines of code that they contain.

A few scripts deal with figure control, both how to place the figure window on the screen and how to arrange the characteristics of the figure when it is printed on the page. I have a procedure for organizing my windows which I find very useful: relatively small Matlab

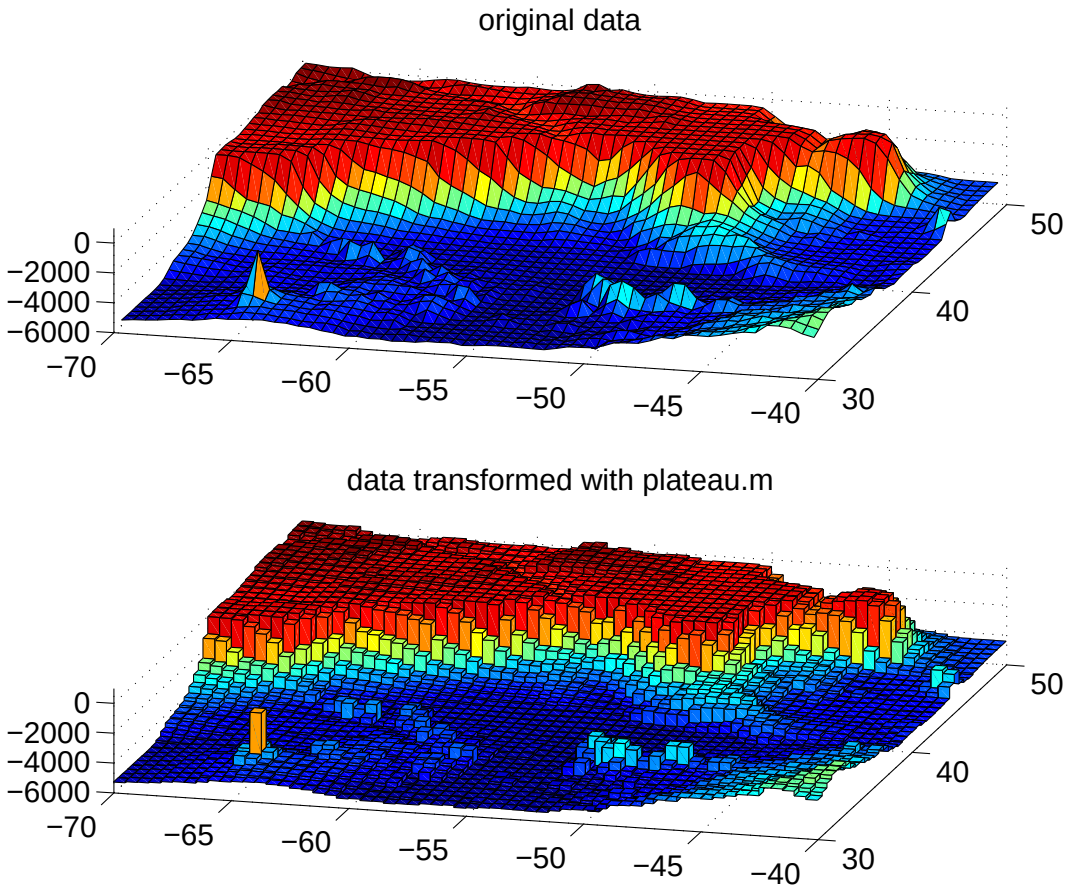

Figure 7: North Atlantic topography to illustrate use of **plateau** script (**plateaudemo**).

window in one corner of my display screen, an editor window in the other screen, and Matlab figure windows in the remaining corners. This way I can see the script, the command line, and the output all at once. I've noticed that many students who don't do this seem to have to rearrange all their windows every time they do anything. For myself, trying to move things with the mouse or other pointing device is usually much more timeconsuming than typing a line of text, so I've created a few Matlab commands to manipulate the windows. I use **newfig** to create a new graphics window; this includes parameters for which window to open, where to put it on the screen, and how big to make it. For instance, **newfig(2,4,'long')** opens figure window (2), puts it in the upper right hand of the screen, and gives it a relatively long height relative to its width. The window can be enlarged with **bigger**, shrunk with **smaller**, and, if covered by another window, can be made visible with **top**.

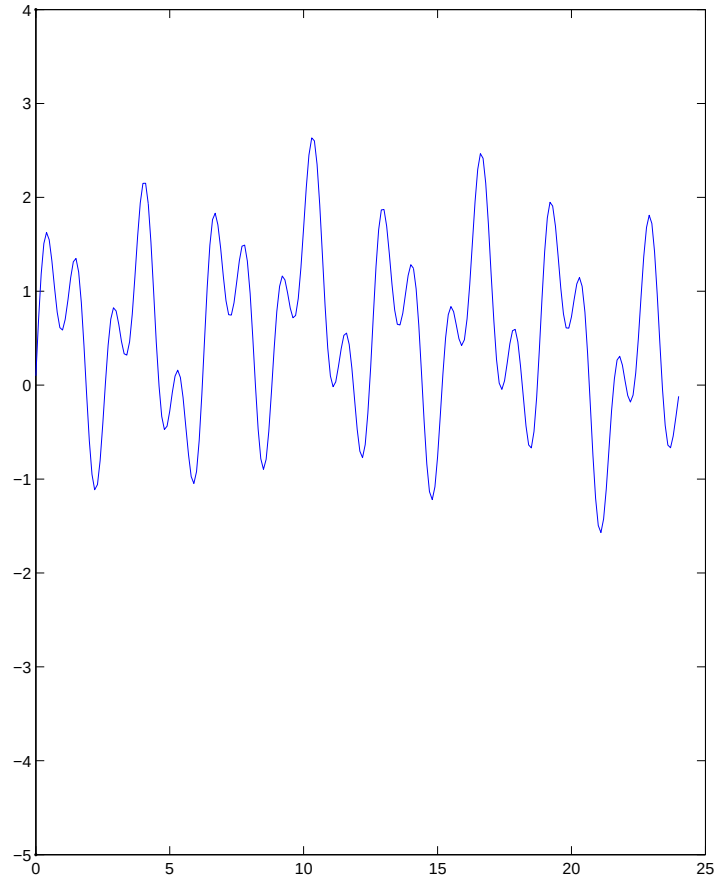

Figure 8: Illustration of a difficult-to-read figure (**wrongright**, option 1).

Setting the size of the rectangle where all graphics will be printed (on a given page) can be done with **paper** for portrait orientation or **landscape** for landscape orientation. The Matlab default has rather wide margins at the top and the bottom, so I often use **paper([1 1])** to give 1 inch margins everywhere.

I've written a number of very simple scripts to faciliate various graphical good practices. Figure 8 shows a graph which wastes a lot of space with no information, has a curve that is hard to see because the (default) linewidth is so small, and uses a small font size that is also hard to see. Students will often create a stack of pages or files with such graphs; when these are viewed on the computer screen the images must be shrunk down to see them all at once, but then the labels are nearly invisible. Also it's hard to get quantitative information from the graph; even seeing where the curve is positive or negative takes some work. Figure 9 shows an improved version. More than one graph is placed on the page. The script **thickline** controls the width of the curve, **marker** (not used here) controls the size of plotting symbols

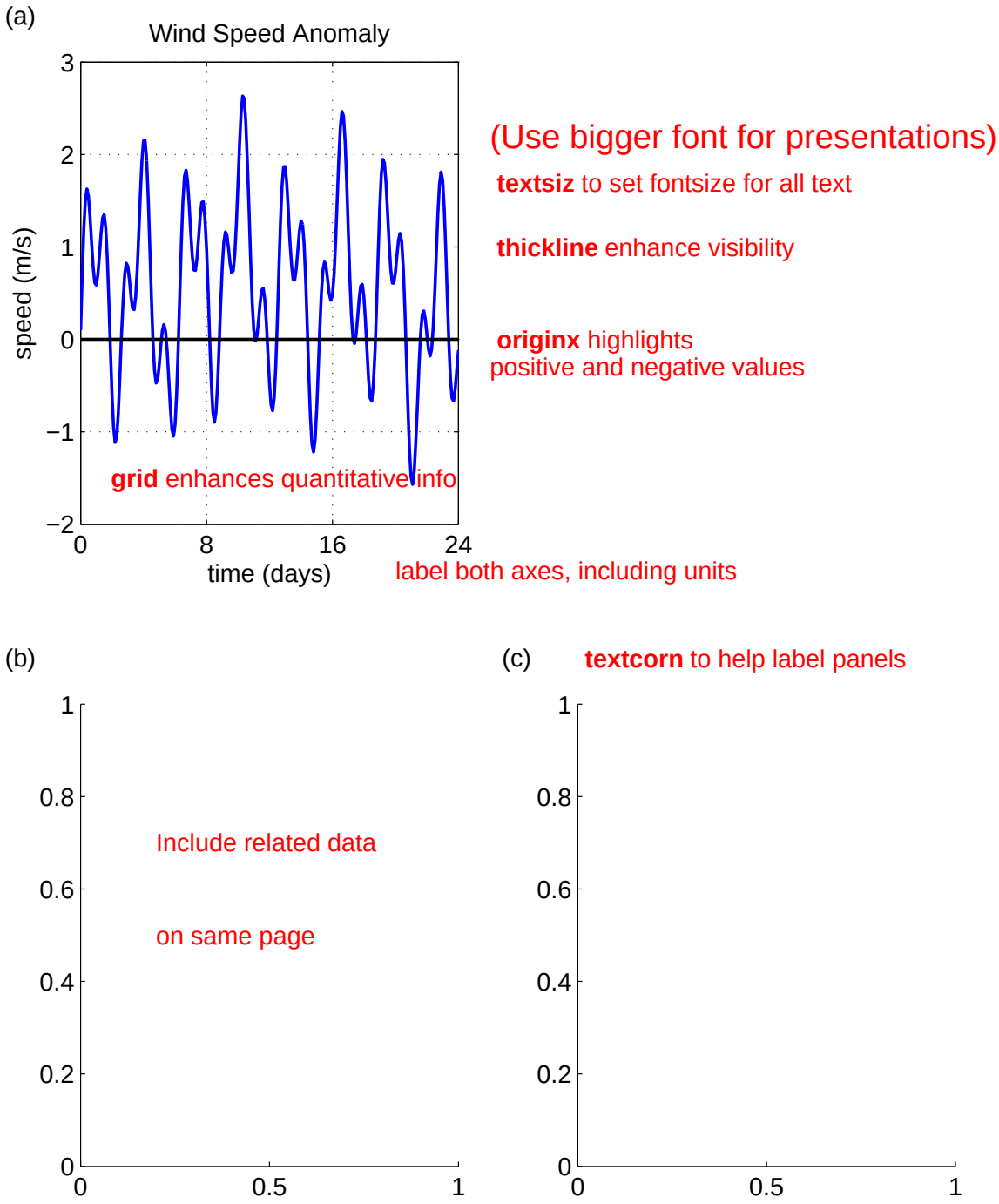

Figure 9: Improved version of Fig. 8, illustrating scripts **xaxis**, **textsiz**, and **thickline** (**wrongright**, option 1).

such as circles or squares, **textsiz** controls the text size, and **originx** places a line on the x axis. Standard Matlab commands also allow for other good practices, such as axis labels, title, and a grid to make it easier to estimate quantitative information from the graph. My script **overgrid** (not used here) gives control over some grid properties and allows the grid to be placed over graphs that use the **pcolor** Matlab command. The scripts **xaxis** and **yaxis** change the limits of only one axis of the graph without the programmer needing to think about what the limits of the other axis are. **textcorn** is useful for putting panel labels (such as (a), (b), etc) on the graph.

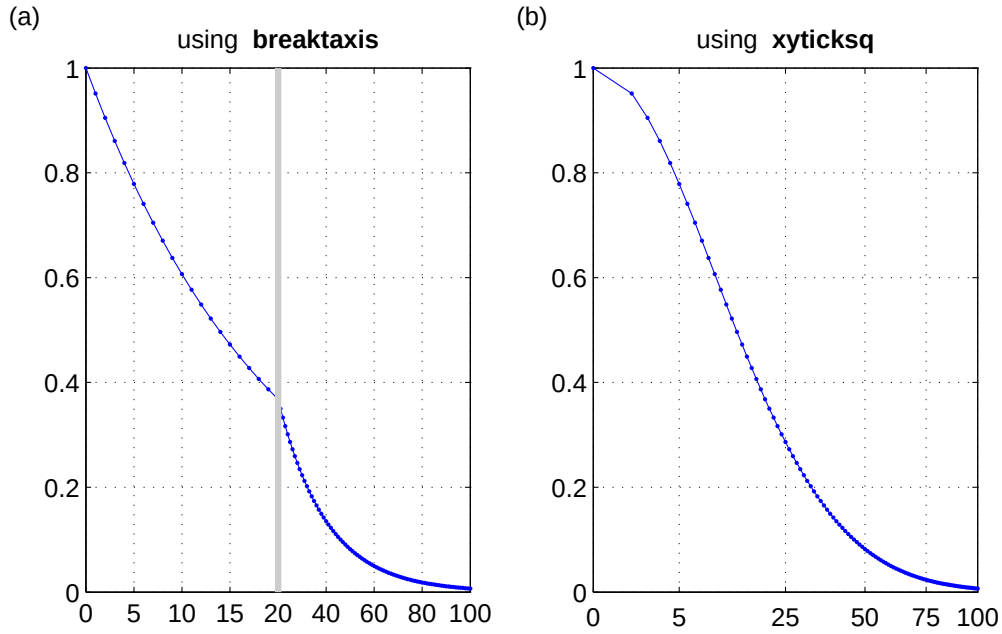

Figure 10: Scripts to help use nonlinear scale using (a) two different scales or (b) distance proportional to square root of variable (**breaktaxisdemo**).

While its sometimes ill-advised to put only a single graph on a page, it is often even worse to put too many on a single page. Are you *sure* you need a single page with all 12 global maps of pressure anomalies superimposed on sea surface temperature? Such a graph *may* be okay when viewed as a PDF on a computer screen, where the viewer can zoom in to various panels at will. It can be quite deadly when used as a slide for a seminar.

Sometimes it is convenient to have one or more axes have a nonlinear scale. There are Matlab functions for plotting on a log scale, but I've found some other scales also useful when a lot more is happening in one small part of the domain than in the rest. This often occurs in the depth coordinate (top few hundred meters of the ocean have more change than bottom few kilometers) or sometimes in the time coordinate. One solution is to use two different scales

(Figure 10a), using **breaktaxis** to define nonlinear variable and tickmark labels. Another is to plot the square root of the axis variable (Figure 10b), using **xyticksq** to label the axis appropriately. This script was also used to make the nonlinear horizontal *and* vertical axes in Figs. 1, 2, and 3.

5 Map Projections

There are many different ways to project part of a sphere on to a plane. I've only implemented azimuthal equal area and some polar projections. "Azimuthal" means that the equator maps to a straight line (none of the other latitude circles do), and "equal area" means that two regions that have the same area on a sphere have the same area when projected on to the plane. Equal area projections are useful because they don't exaggerate the size of some features; the main drawback of such projections is that they don't preserve angles. For instance, two vectors that are at right angles to each other on the sphere will not necessarily be at right angles in the projection.

The main building block of all the routines is **projea**, which transforms any set of (latitude,longitude) locations into a set of (x,y) positions on a plane using an azimuthal equal area projection. The script **projquiv** projects a vector field; **projquiv** doesn't contain any graphics, but returns (x,y) and (u,v) values which can then be graphed with **quiver** or the other vector plotting routines. To make contour plots, the original (lon,lat) grid vectors (or 2D arrays) can be converted to (x,y) arrays which can then be used with contouring scripts such as **contour.m** or **contourfP.m**. Finally, **glatlon** plots a grid of latitude and longitude circles on to the graph. Figure 11 shows an example using these scripts.

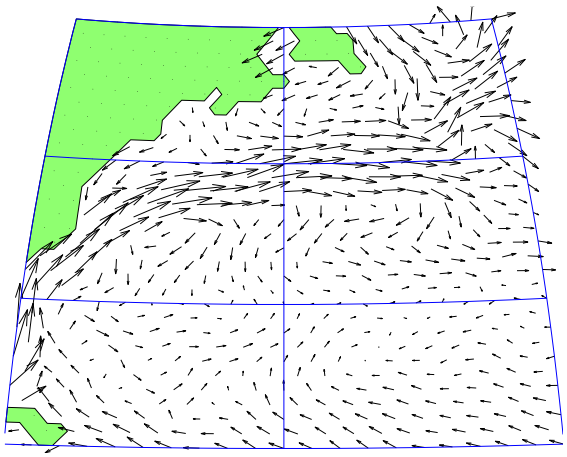

Figure 11: Contour and quiver plots using an azimuthal equal area projection with same data as in Fig. 6 (**projdemo**).

I've also included a script to project global data. The projection used in **projea** looks increasingly distorted as the longitude range becomes bigger, and for a longitude range of 360 degrees the script fails. Therefore I plot global data on a series of smaller sectors of a sphere. One complication with global data is that the western and eastern "edges" of the

data (usually either 180^o longitude or 0^o longitude) will usually fall inside some ocean—not a good way to display ocean data. Mscript **loncycle** solves this problem by shifting the coordinates to whatever boundaries are desired. The global projection routines for contour plots and quiver plots are **contglobe** and **quivglobe** (Fig. 12). Note that **contglobe** calls **contourfP.m**. The global contouring in Fig. 12 is done with a single call to **contglobeP**:

```
contglobeP(lon,lat,D/1000,ci,'w-','bg',0,'none',.5,xlohi);
```

The parameters passed to **contglobeP** in the example above tell the routine to use a solid white lat-lon grid, a color palette with blues for negative values and greens for positive values, no contour lines, and color ranges modified by the array **xlohi**. I experimented with combining the Atlantic and Indian in a single projection, but the format shown here produces less distortion around the edges, at the cost of having more edges. Longitude bands at the edges of the sectors are included in neighboring sectors. The demonstration script for these functions, **globedemo.m**, must be run in a directory with files **bathdeg.mat** and **surfvelclim.mat**.

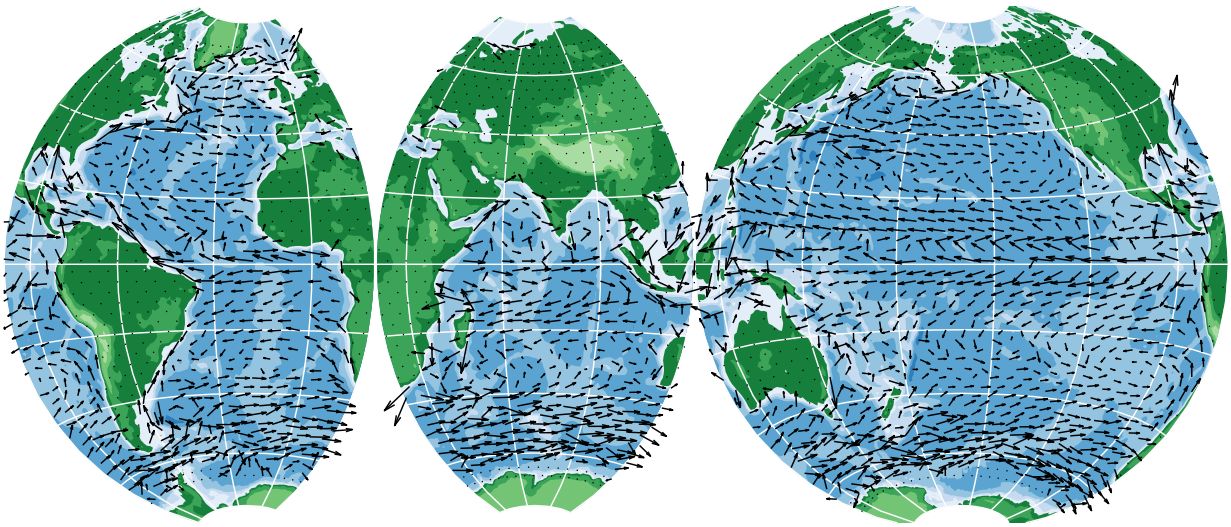

Figure 12: Global contour and quiver plots (topography and surface velocity) using an azimuthal equal area projection (**globdemo**, which caclls **contglobe.m** and **quivglobe.m**).

Polar projections are illustrated in Fig. 13 and can be used for creating contours (**projpol.m**), velocity vectors (**poluvc.m**), and latitude-longitude grids (**polgrid.m**). The scripts choose whether to project around the north pole or south pole based on the range of latitudes passed to the script. The polar projection scripts give three options for projections. A conceptually simple projection is the "view from infinity" which is to project the globe

into the equatorial plane (Fig. 13, option 2). This introduces fairly substantial distortions equatorward of 45^o and is not an equal-area projection. The equal area-projection is defined by the criterion that a ring between latitude ϕ and $\phi + d\phi$ on the Earth has the same area as the ring between radius r and $r + dr$ in the projection plane. It is represented by

$$r\ dr = -\cos(\phi)\ d\phi \tag{1}$$

where there is a negative sign because increasing ϕ corresponds to decreasing r. This projection is option 3 (Fig. 13). Integrating this equation gives us the relationship between latitude and radial distance on the projection:

$$r = \sqrt{2}\sqrt{1 - \sin(\phi)}. \tag{2}$$

The simplest option is to make r proportional to the latitude distance (on a sphere) from the pole (Fig. 13, option 1). As the figure shows, option 3 is fairly similar to the equal-area option, and all three options give fairly similar results within 45^o of the pole. Therefore I usually use option 1 which is easy to describe and interpret and which is relatively close to being an equal-area projection.

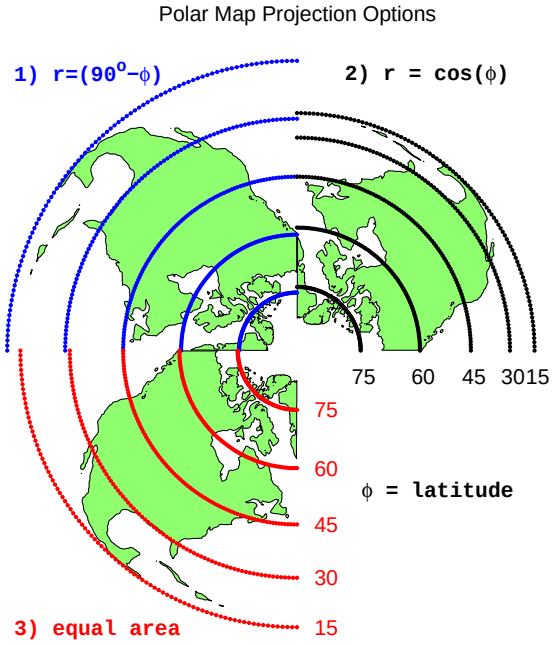

Figure 13: Three polar projections showing latitude circles and northern hemisphere topography.