

1 Adjust the axes to show *longitude* and *latitude* on your map

Let us use the Matlab *script* that we made earlier and saved as `f1.m`. In the version that we give you, I have annotated each of the lines a little so that you can remember what it does.

Now, simply type the name of the *file* (without the extension) in the Matlab prompt, and you'll get a picture with the axes showing, simply, the *pixel* numbers of the map.

Note: if you copy and paste the lines below from within your PDF viewer, some quotes (') may not copy right. Add those quotes by hand, or copy them from the program files `f1.m` through `f8.m`.

```
>>f1
```

Let's be explicit about what the axes are showing, even though we will change it in a minute.

```
>>xl=xlabel('easting (pixel number)');  
>>yl=ylabel('northing (pixel number)');
```

Now, we get the data range of the easting, in decimal degrees of longitude. And then the same for the northing, in latitude. We *know* what the relevant variables and their id numbers are since these were contained in the variables `VarName` and `VarIDs`, which you can inspect again to be sure.

```
>>xr=netcdf.getVar(ncid,0);  
>>yr=netcdf.getVar(ncid,1);
```

We want to supply this information to the map-making command to get something that is properly showing longitudes and latitudes. Note that the (first row, first column) element is at longitude `xr(1)` and latitude `yr(2)`, and that the (last row, last column) element is at longitude `xr(2)` and latitude `yr(1)`. We need to do `axis xy` to tell Matlab that we get the larger latitudes up above, as otherwise, the default for `imagesc` is `axis ij`, which interprets the data as an indexed *matrix* instead of as a Cartesian object. Check `help imagesc` for the conventions and *look* at your output!

```
>>imagesc(xr,yr([2 1]),zr)  
>>axis xy
```

Now we have the map properly geographically referenced, let us relabel the axes also:

```
>>xl=xlabel('longitude (degrees)');  
>>yl=ylabel('latitude (degrees)');
```

And finally, end by restricting the range and equalizing the axis stretchings again, and issue the `print` command. One unit on the horizontal axis with one unit on the vertical axis will form a little *square*. This behavior pertains to the visual appearance only, and the nearer to the poles your area is, the less appropriate this choice will be. In reality lines of longitude converge to the poles.

```
>>axis image  
>>print -dpng mynewtopo2
```

Save everything we did in this example into a Matlab program file, let's call it `f2.m`.

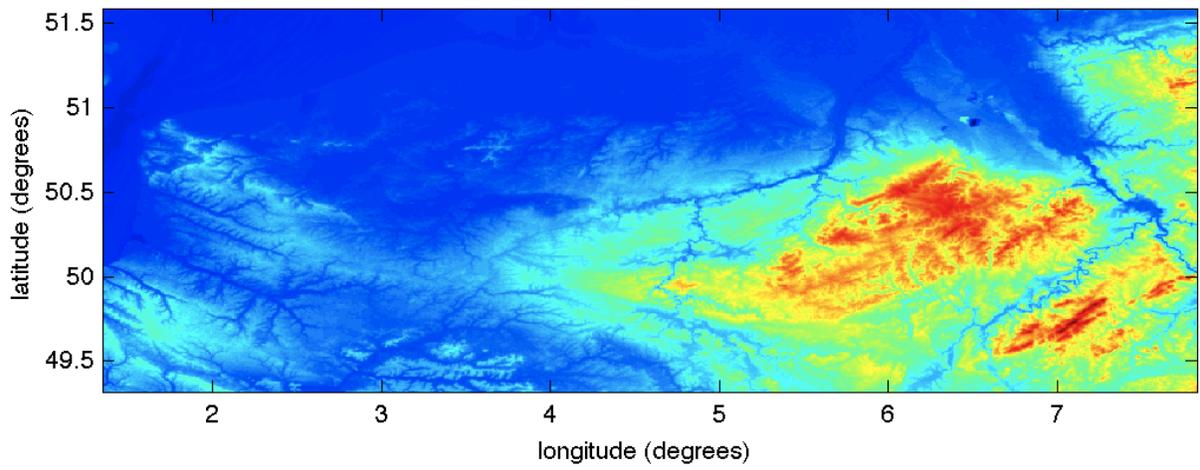


Figure 1: This figure is the result of the material in this section. It's what you get from f2.m.

2 Draw a straight-line *profile* through your map

You can just go on with the exercise, but if you had quit Matlab and restarted it just now, you can execute `f2.m` to get back in business. In this section we will be making `f3.m`.

```
>>f2
```

Actually, to prepare *two* figure panels, let's issue a `subplot` command first, where we will assign a *handle*, which I call `ah(1)`, to the *first* panel (last argument to `subplot`), of a figure that will have panels in *two rows* (first argument to `subplot`) and *one column* (second argument to `subplot`).

```
>>ah(1)=subplot(2,1,1);  
>>f1
```

You see that we are back in *pixel* space, which is how Matlab sees the data anyway, and which is easier for you if you want to pick out specific rows or columns. We'll pick pixels between which to draw profiles, and then readjust the plot axes to show longitude and latitude again, later.

Open the second figure panel, sticking to the overall layout, and assign a handle to it:

```
>>ah(2)=subplot(2,1,2);
```

Addressing a *row*, taking all columns, e.g. the 300th row, goes as follows. Remember the matrix variable containing the plotted data is called `zr`. Thus, we extract the 300th row and assign it to

```
>>pr1=zr(300,:);
```

If you plot this variable, you'll see how the topography varies as a function of the horizontal variable.

```
>>plot(pr1)
```

But let's be smarter than that, and assign the correct *longitudes* to this profile. We know what they were: they go between `xr(1)` and `xr(2)`, and there are exactly `zdim(1)` of them. However, we still need to *generate* the proper longitudes, as many as we have data values in the profile, and they are linearly spaced. Then, plot the result.

```
>>lon1=linspace(xr(1),xr(2),zdim(1));  
>>plot(lon1,pr1)
```

Now it looks properly referenced! Finish up by trimming the axes to the data range, put on labels.

```
>>axis tight  
>>xl(2)=xlabel('longitude (degrees)');  
>>yl(2)=ylabel('elevation (m)');
```

Make the *top* panel active again, replot the data in the top panel in longitude/latitude coordinates.

```
>>axes(ah(1))  
>>imagesc(xr,yr([2 1]),zr); axis xy  
>>xl(1)=xlabel('longitude (degrees)');  
>>yl(1)=ylabel('latitude (degrees)');
```

Save everything we did in this example into a Matlab program file, let's call it `f3.m`.

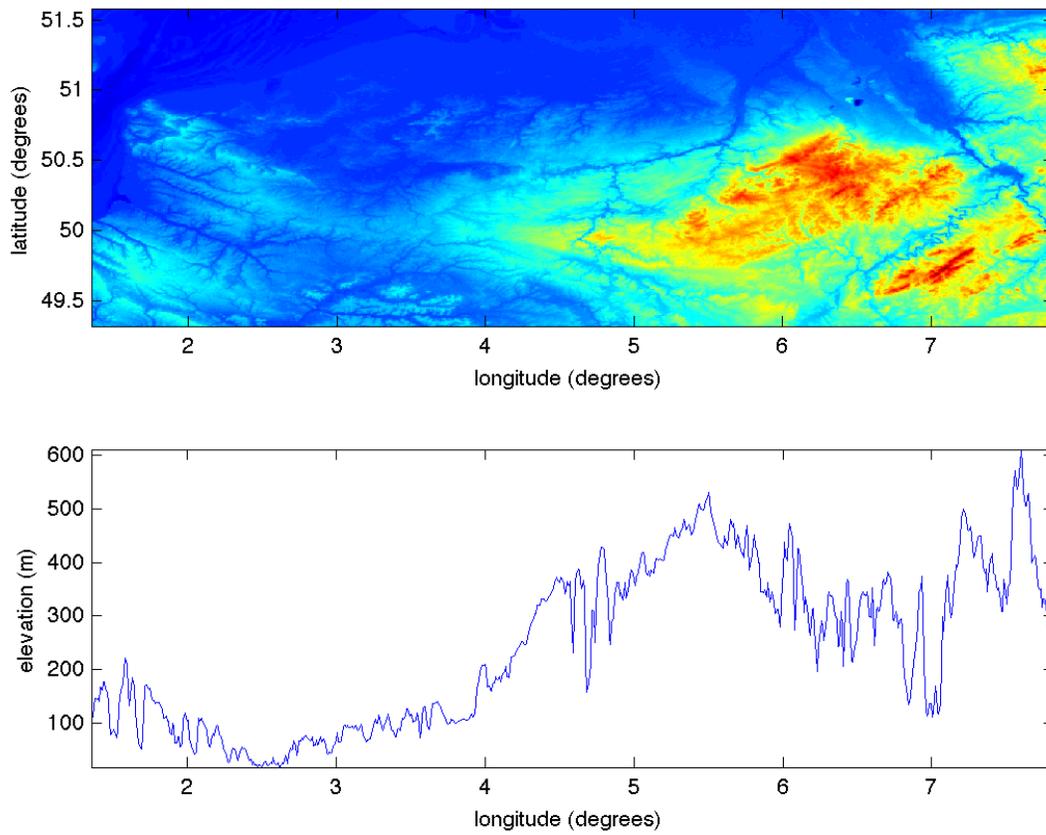


Figure 2: This figure is the result of the material in this section. It's what you get from f3.m.

3 *Identify* a straight-line *profile* through your map

Clearly what's missing from the previous section is for the *top* panel to identify *where* the profile was taken. We should be plotting a line at the appropriate location *on top* of the top panel, to identify the location of the profile. In this section we will be making `f4.m`.

To make it more interesting, I will switch to a profile along the *vertical* direction. Again, start with `f2` and `f1` to recoup the variables if you lost them, and for a visual of which pixel it is that you want profiled, and open the subpanels.

```
>>f2; ah(1)=subplot(2,1,1); f1; ah(2)=subplot(2,1,2);
```

Looks to me like somewhere around pixel 550 it gets interesting, so extract that profile, assign it to a new variable, and produce the appropriate latitudes for it! Same as in the previous section, *mutatis mutandis* of course: longitudes become latitudes, first dimensions become second dimensions, and so on. I'm even switching the axes around so you don't get confused.

```
>>pr2=zr(:,550);
>>lat2=linspace(yr(2),yr(1),zdim(2));
>>ah(2)=subplot(2,1,2);
>>plot(pr2,lat2)
>>axis tight
>>xl(2)=xlabel('elevation (m)');
>>yl(2)=ylabel('latitude (degrees)');
```

At this point, repeat the steps from the previous section to put the map on top, properly referenced.

```
>>axes(ah(1))
>>imagesc(xr,yr([2 1]),zr)
>>axis xy
>>xl(1)=xlabel('longitude (degrees)');
>>yl(1)=ylabel('latitude (degrees)');
```

And then keep this top panel *active* so that all further plot commands go right on top:

```
>>hold on
```

But we need to still figure out at which longitude pixel number 550 was located, so we can plot a line identifying it! We have all the pieces in hand to be able to find it. It is a matter of simply finding what the 550th column out of the total number of columns is (we get this from `zdim(1)`), as a proportion of the total longitudinal interval (which we get from `xr`). But before we do that we need to realize that the data format has the variable `zdim` as an *integer*, which we need to convert to a *float* to be able to do any computation with it. Using Matlab's function `double`:

```
>>lonpr2=xr(1)+(550-1)/double(zdim(1)-1)*(xr(2)-xr(1));
```

You might have to write this down on a piece of paper to properly understand it. But then you should be able to do the equivalent thing for when the profile is longitudinal, and you need to figure out the proper longitude! All that is left is to add the line on to the map.

```
>>plot([lonpr2 lonpr2],yr,'LineWidth',2,'Color','k')
>>hold off
```

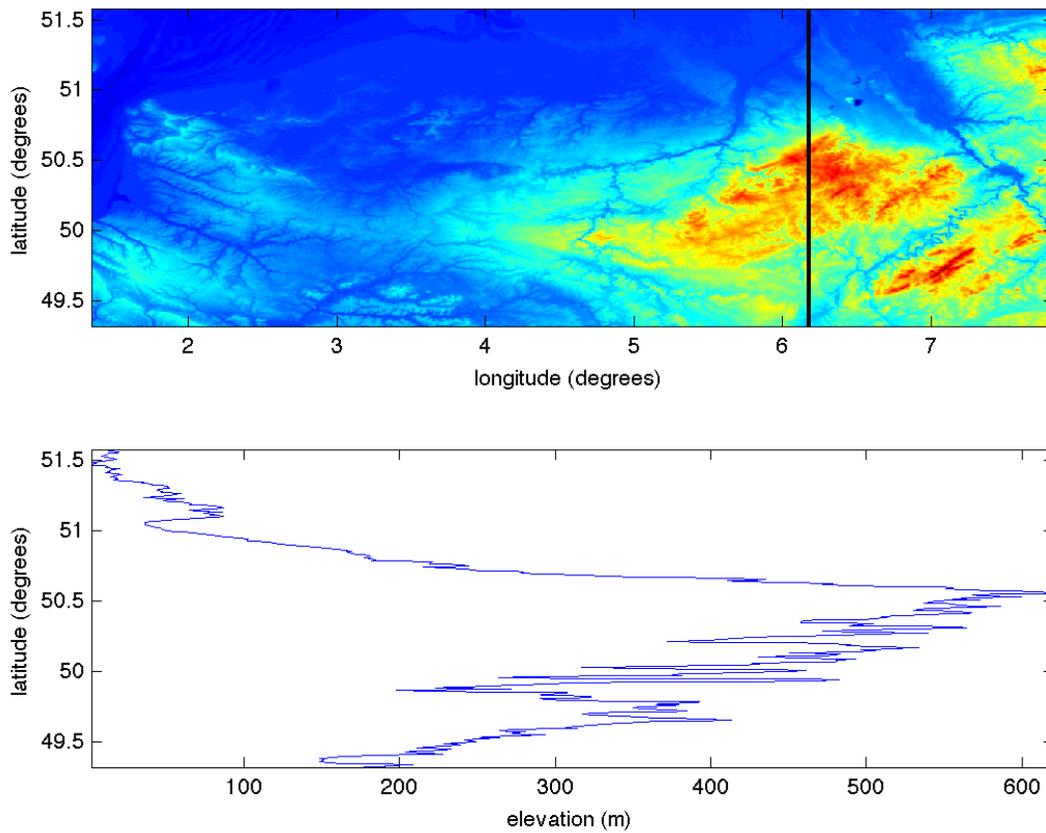


Figure 3: This figure is the result of the material in this section. It's what you get from f4.m.

4 *Isolate and analyze* a polygonal piece out of your map

Here you will learn how to draw a polygon, free-hand, plot it on your map, and then compute statistics of the numbers that Matlab finds for you inside that polygon, or out. We make `f5.m`. We begin once again by calling the pixel-referenced map, after *clearing* the figure.

```
>>clf
>>f1
```

To get *crosshairs* and draw a freehand polygon, type

```
>>[pg1,pg2]=ginput
```

and click away, ending with *enter*. When I was done, I had obtained the result:

```
pg1 =
    305.3191
    520.9850
    692.4988
    549.8537
```

```
pg2 =
    280.2661
    394.0426
    225.9251
    134.2247
```

I can now plot the polygon on the map! Note that I *close* it!

```
>>hold on
>>plot(pg1([1:end 1]),pg2([1:end 1]),'LineW',2,'Color','k')
>>hold off
```

And now I want to figure out what the mean value, the variance are, and so on, of the values that lie *within* the polygon. This isn't particularly complicated: Matlab has a function for it, `inpolygon`. In the *pixel* domain, I first have to tell Matlab what the column numbers and the row numbers are that I *have*:

```
>>xrall=double(1:zdim(1));
>>yra11=double(1:zdim(2));
```

Then I need to turn this into a *grid* with a pair of (row,column) numbers for *each* of the points in my dataset. For this, we use the function `meshgrid`.

```
>>[xrall2,yrall2]=meshgrid(xrall,yra11);
```

And then I need to use the `inpolygon` function to return *logical* indices to all of those entries in the matrix `zr` that fall inside of my polygon:

```
>>inside=inpolygon(xrall2,yrall2,pg1,pg2);
```

At this point I would like to check that I've gotten it right. The best way is to generate a copy of the data set, set all the values inside the polygon to 1 and all the values outside (using the *negation* operator \sim) to 0, and take a look.

```
>>zr2=zr;
>>zr2(inside)=1;
>>zr2(~inside)=0;
>>imagesc(zr2)
>>axis image
```

After verifying that I have indeed properly isolated the polygon of interest, I can simply operate on the original matrix, using the logical array variable `inside` that I just created, with the functions `mean` and `var`. Should there be any NaN values inside, I should use `nanmean` and `nanvar`. By not closing the following statements with a semicolon, the values will be reported to the prompt. Before I do so I will plot the map again using `f1`, and plot the polygon on top again.

```
>>f1
>>hold on
>>plot(pg1([1:end 1]),pg2([1:end 1]),'LineW',2,'Color','k')
>>hold off
>>mean(zr(inside))
>>var(zr(inside))
```

For me, the results for the *mean* and *variance* were:

```
ans =

    362.7773

ans =

    1.2599e+04
```

Can you rewrite this function in *geographical* instead of in *pixel* coordinates? If all you want is the plot at the end, you'd start from `f2` and then tweak the material in this and previous section to overlay the polygon in longitude and latitude instead of in column and row (pixel) numbers.

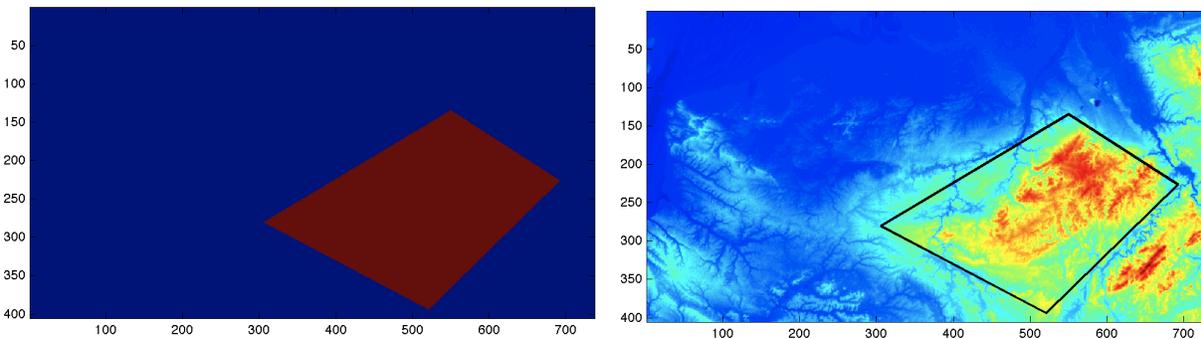


Figure 4: This figure is the result of the material in this section. It's what you get from `f5.m`. On the *left* we identify the polygon explicitly, on the *right* we show the map with the polygon on top.

5 Revisiting the *histogram*

A *histogram* finds the number of times a certain range of values occurs, within the edges of a set of predefined bins. If you had millions of observations, and you could set the bin widths to something infinitesimally small, and you *normalized* the area under the histogram to be exactly one, you'd be getting something that reflects the statistical *distribution* by approximating the *probability density function*. Matlab's `hist` function is very flexible. Let's run `f1` to get our variables back in place, and then make some histograms with varying numbers of bins. We will be making `f6.m`.

Note that the vertical axis is in unnormalized *counts*, the number of *pixels* that have a value within the bin range. With this type of data, such an accounting is only a crude approximation of a proper *hypsometry*, which should weight the occurrences of certain elevations by their *area*. Here, every pixel does not represent the same area, yet every one of them counts equally towards the total.

```
>>f1
>>ah(1)=subplot(2,3,1); hist(zr(:),6)
>>ah(2)=subplot(2,3,2); hist(zr(:),12)
>>ah(3)=subplot(2,3,3); hist(zr(:),24)
```

I'm now going to address all of the axis handles at the same time and set the limits of the horizontal axis to the minima and maxima of the data set under consideration.

```
>>set(ah,'xlim',[min(zr(:)) max(zr(:))])
```

While I'm at it, I realize it really would be nice if the areas under every one of the histograms were normalized to unity. To accomplish this, I will use a version of `hist` that returns *output*, which I then need to fiddle with and plot myself, using `bar`. I'll do this on the second row, and after that, it's a pretty safe bet that I will be able to also equalize the axis range in the vertical dimension.

```
>>ah(4)=subplot(2,3,4);
>>[a,b]=hist(zr(:),6); br(1)=bar(b,a/sum(a)/[b(2)-b(1)],1);
>>ah(5)=subplot(2,3,5);
>>[a,b]=hist(zr(:),12); br(2)=bar(b,a/sum(a)/[b(2)-b(1)],1);
>>ah(6)=subplot(2,3,6);
>>[a,b]=hist(zr(:),24); br(3)=bar(b,a/sum(a)/[b(2)-b(1)],1);
>>set(ah(4:6),'xlim',[min(zr(:)) max(zr(:))],'ylim',[0 1/(max(zr(:))-min(zr(:)))]*5)
```

Piece together the arguments to `bar` (center, height, width) by consulting the help — and changing my choices. It is easy to verify that the area under these histograms is exactly one. For this reason, I can use Matlab's statistical functions to plot *reference distributions* right on top of the lower row of panels. I can *tell* these distributions aren't normal (and so can you!), but if we compare them quantitatively, we'll get an even better idea.

```
>>dxr=linspace(min(zr(:)),max(zr(:)),100);
>>dyr=normpdf(dxr,mean(zr(:)),std(zr(:)));
>>for index=4:6
>> axes(ah(index)); hold on; plot(dxr,dyr,'LineW',2,'Color','r')
>>end
```

Note that I'm starting to take real advantage of the axis *handles* to address them individually as part of a `for-end` loop!

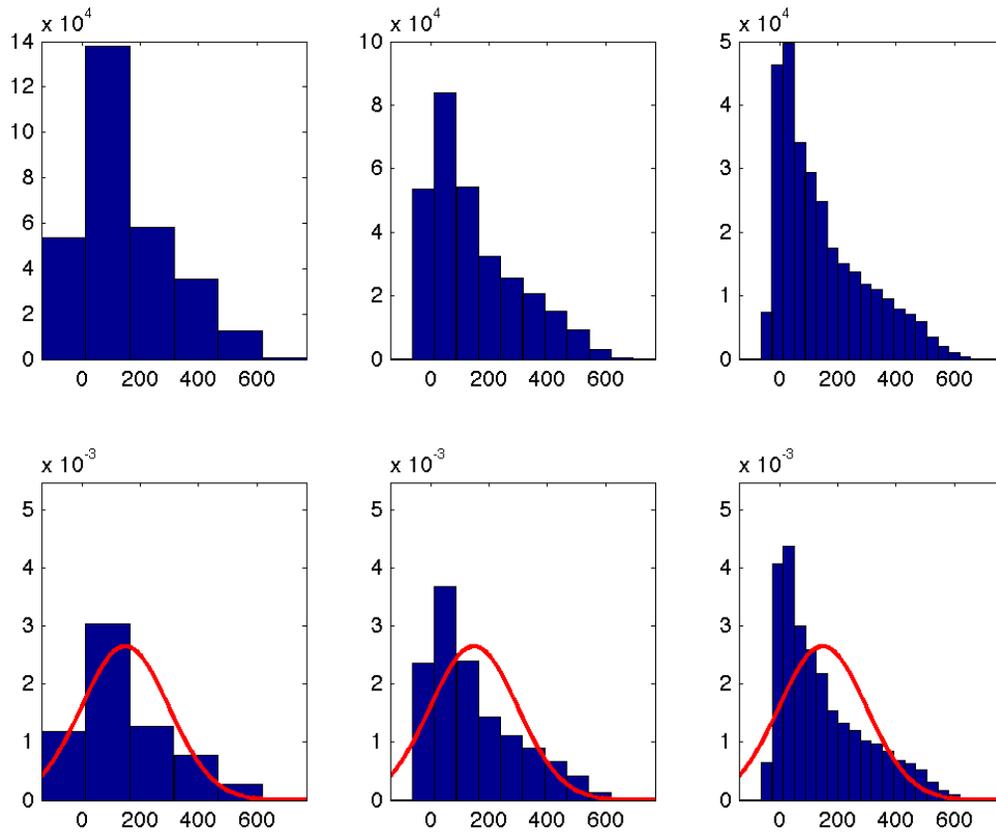


Figure 5: This figure is the result of the material in this section. It's what you get from `f6.m`. Can you now add the labels for the horizontal and vertical axes?

6 Looking at topographic *slopes*

As the last bit in this mini-series, let us try to look at how “rough” topography is. A *slope* is a *derivative*, and a derivative is (the limit of) a *difference* that is normalized by the *step size* in the direction in which the difference is being taken. We will be making `f7.m`.

We adapt `f3` by plotting, instead of the simply extracted profile, the first derivative of that profile. We use Matlab’s function `diff`, whereby we note from its `help` page that it computes a *first difference*, which means that it returns *one less point* of output than what you feed it as input. Consequently, we need to quote the longitudes of where the derivative is taken in a manner that is shifted by *half a pixel* compared to where the original profile was quoted.

The only important change (other than that I changed the row number for this example, and added the profile location in the top panel, and of course, adjusted the labels for the vertical axis in the bottom panel) is in the line

```
plot([lon1(2)-lon1(1)]/2+lon1(1:end-1),diff(pr1)/[lon1(2)-lon1(1)]/1000)
```

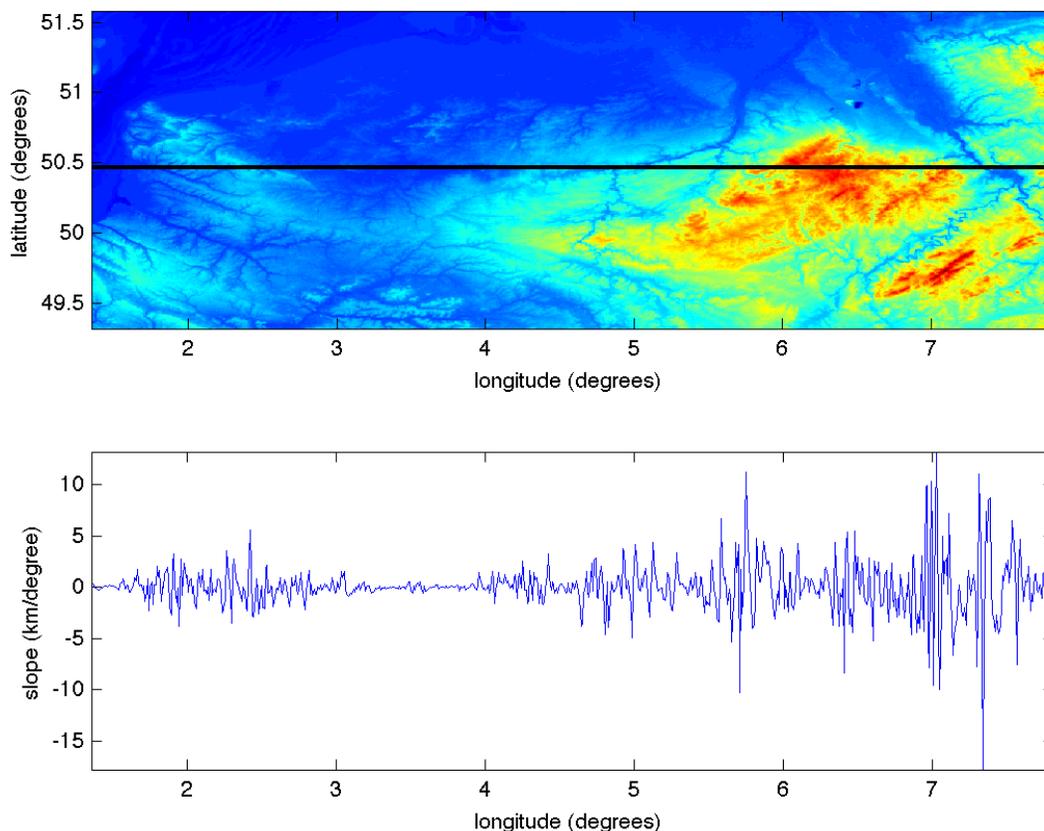


Figure 6: This figure is the result of the material in this section. It’s what you get from `f7.m`.

7 Looking at topographic *roughness*

In the previous section we looked at the derivative in a certain *direction* at a certain profile *location*. Matlab is smart enough to plot the derivative in a certain *dimension* for all the rows (or columns) of the map. Matlab's first is the row dimension (*down* the matrix), its second is the column dimension (*across* the matrix). We will be making `f8.m`.

We adapt `f2.m` quite simply by changing the variable to be plotted from `zr` to `diff(zr,[],1)` for the latitudinal derivative, and to `diff(zr,[],2)` for the longitudinal derivative.

Once again, the changes are minor. We need to compute both the longitudes and the latitudes of our data grid:

```
>>xr=netcdf.getVar(ncid,0);
>>lon1=linspace(xr(1),xr(2),zdim(1));
>>yr=netcdf.getVar(ncid,1);
>>lat2=linspace(yr(2),yr(1),zdim(2));
```

And then we need to compute the derivatives, and plot those in subpanels. To adjust the range of colors being plotted to one that *shows* meaningful variations, we use Matlab's `caxis` command. And finally, we add a `colorbar` and invoke `title` for annotation.

```
>>ah(1)=subplot(3,1,1);
>>imagesc(xr,yr([2 1]),diff(zr,[],2)/[lon1(2)-lon1(1)]/1000)
>>axis image xy; caxis([-3 3]); colorbar
>>t1(1)=title('horizontal (longitudinal) slopes (km/degree)');
>>ah(2)=subplot(3,1,2);
>>imagesc(xr,yr([2 1]),diff(zr,[],1)/[lat2(1)-lat2(2)]/1000)
>>axis image xy; caxis([-3 3]); colorbar
>>t1(2)=title('vertical (latitudinal) slopes (km/degree)');
```

The above two *normalized finite differences* amount to *directional derivatives*. To get an idea of the overall roughness, we simply take the square root of the sum of their squares: in that case we make a measurement of the *magnitude* of the *vector gradient*, which I'll call *roughness*. Since we use `diff` and thereby *lose* a point in the dimension in which it is applied, I simply trim the resulting matrices to have the same size. Matlab's own `gradient` function would have done a slightly better job than I here, but this will do for now.

```
>>ah(3)=subplot(3,1,3);
>>dzdy=diff(zr,[],1)/[lat2(1)-lat2(2)]/1000;
>>dzdx=diff(zr,[],2)/[lon1(2)-lon1(1)]/1000;
>>imagesc(xr,yr([2 1]),sqrt(dzdy(1:end,1:end-1).^2+dzdx(1:end-1,1:end).^2))
>>axis image xy; caxis([0 3]); colorbar
>>t1(3)=title('roughness (km/degree)');
```

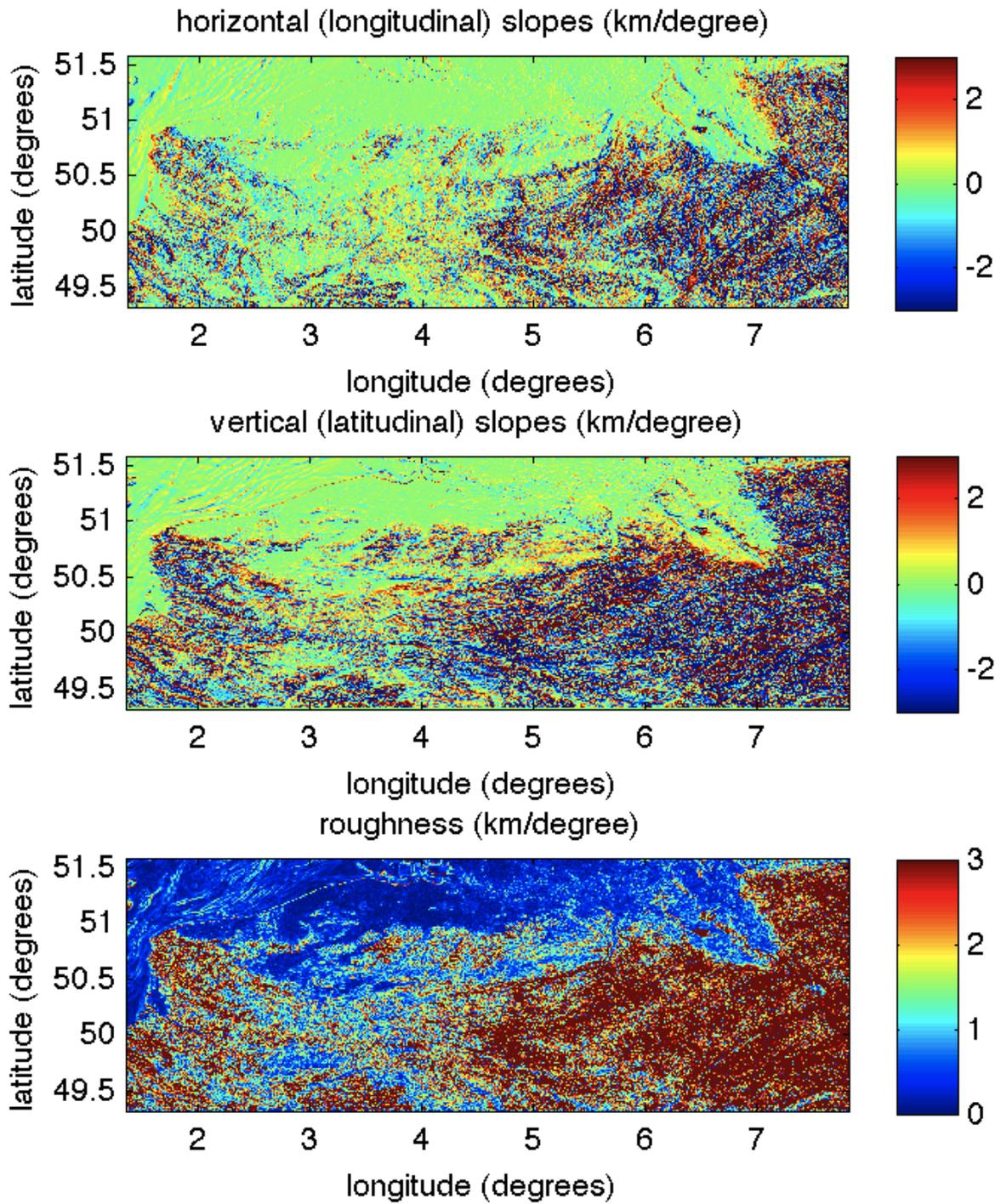


Figure 7: This figure is the result of the material in this section. It's what you get from f8.m.