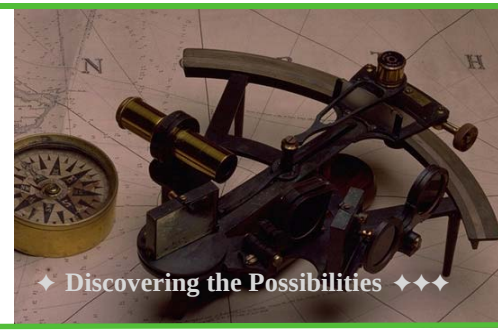
Chapter 5



Creating Contour Plots

Basic Contour Plots

Nearly everyone is familiar with what a contour plot is from looking at topographical maps. A contour plot is a way of representing a three-dimensional surface on a flat, two-dimensional surface. The third dimension is represented by contour lines, usually drawn, at least on topographical maps, at some equally-spaced contour interval.

Almost any two-dimensional data set can be contoured in IDL. The values of the two-dimensional data will represent the "height" or third dimension to be contoured. To learn how contour plots are created, we will load a simple 2D array with the cgDemoData command, a Coyote Library routine you downloaded to use with this book. You can use the *Help* command to determine that this is a 41 x 41 floating point array. The *Min* and *Max* commands allow you to determine the array's data range, which in this case is 0 to 1550.

```
IDL> data2D = cgDemoData(2)
IDL> Help, data2D
     DATA2D          FLOAT     = Array[41, 41]
IDL> Print, Min(data2D), Max(data2D)
     0.000000      1550.00
```

A contour plot in its most basic form can be created just by passing this array as the argument of the IDL traditional graphics command *Contour*, like this.

```
IDL> Contour, data2D
```

You see the result in Figure 1. This is a PostScript rendition of the contour plot. On the display, this is rendered as white lines on a black background.

(We will have more to say about colors in just a moment.) Not terribly impressive, I admit. But, wait, it gets better.

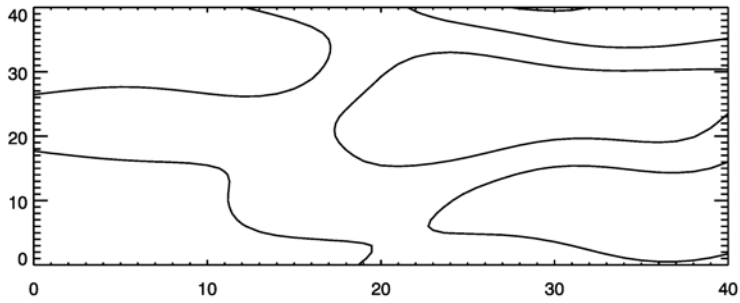


Figure 1: The most basic contour plot in IDL. There are 60+ keywords that you can use to enhance the information content of the Contour command to make it significantly more valuable to you.

There are more than 60 different keywords you can use with the *Contour* command to make the contour plot significantly more valuable to you. Before we start to talk about the 15-20 of those keywords that you will absolutely want to know how to use to add more features to a contour plot, notice what is already there. The axes are labeled!

You will notice that both the horizontal or X axis, and the vertical or Y axis have ranges that extend from 0 to 40. This should remind you of what happens with the *Plot* command when you only pass the dependent data set to the command. IDL creates the independent data to plot the dependent data against.

The same thing is going on here, except that IDL needs to create both an X and a Y vector that specify the locations of each value in the 2D data array. By default, these two vectors are just index vectors that are the same size as the dimensions of the data array. In other words, IDL essentially executes these commands, when the *Contour* command is called as it was in the statement above.

```
IDL> s = Size(data2D, /Dimensions)
IDL> xvec = IndGen(s[0])
IDL> yvec = IndGen(s[1])
IDL> Contour, data2D, xvec, yvec
```

The X and Y data vectors must be monotonically increasing (or decreasing) in value, but they do *not* have to be regularly spaced. The X and Y data

parameters also don't have to be vectors. They can be 2D arrays with the same dimensions as the data array. Each element of the X and Y arrays will then locate the corresponding element in the data array.

Normally, the X and Y data vectors represent some physical property of the data or the way it was collected. For example, X and Y might represent the longitude and latitude locations of the data, or they might represent physical properties like temperature and pressure.

Let's assume we are contouring satellite data and the X and Y vectors represent longitude and latitude locations. We want to express them, of course, in map units. Normally, for these kinds of map contours, the X and Y vectors would be expressed in terms of projected meters. If we assume this data represents, say, atmospheric pressure collected by satellite and gridded to 25 km grid cells, centered over Boulder, Colorado, USA, in an orthographic map projection, then the projected meter values we are talking about extend from -512500 to +512500 in both X and Y. (Don't get too hung up on these details. I have some points I want to make about contour plots, and sometimes it is easier to make those points using imagination rather than reality. You will have plenty of time to use the techniques I am going to show you with messy real data.)

If I want my data range to go from -512500 to +512500, then the center of the first grid cell will be at -500000 and the center of the last grid cell will be at +500000. I can make my vectors like this.

```
IDL> s = Size(data2D, /Dimensions)
IDL> lon = IndGen(s[0]) * 25000L - 500000L
IDL> lat = IndGen(s[1]) * 25000L - 500000L
IDL> Print, Min(lon), Max(lon)
     -500000      500000
IDL> Print, Min(lat), Max(lat)
     -500000      500000
```

These are large values to display on an axis, so it would be better to divide these by, say, 1,000 and indicate this in the plot annotation. We could write code like this.

```
IDL> lon = lon / 10^3
IDL> lat = lat / 10^3
IDL> xtitle = 'Longitude (projected meters x 1000)'
IDL> ytitle = 'Latitude (projected meters x 1000)'
IDL> Contour, data2d, lon, lat, XTitle=xtitle, $
        YTitle=ytitle, Title='Atmospheric Pressure'
```

You see the result in Figure 2.

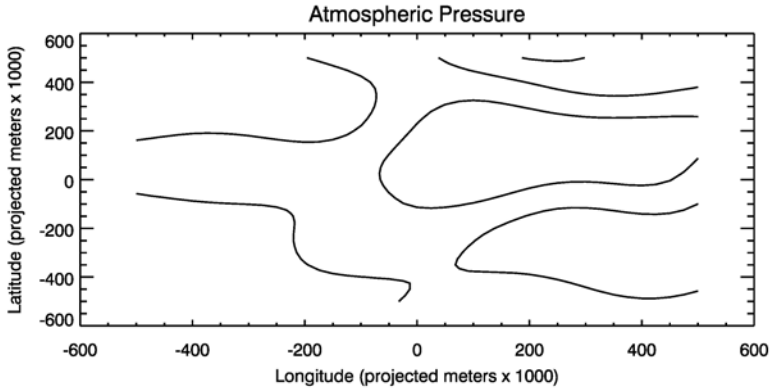


Figure 2: This plot is not a whole better than the first, except that the axes are labeled properly. Or are they? Note our old friend axis auto-scaling rearing its ugly head!

Visually, this contour plot is not much better than the previous plot, but at least the axes are labeled properly. Or, are they? Here we see a *very* common problem with contour plots in IDL that we first encountered with line plots. Namely, the contour axes are auto-scaled. They are setting end points that reflect IDL's aesthetic sensibilities more than they do our desire for a decent looking contour plot. I would say this is a problem in contour plotting about 80 percent of the time.

The solution, of course, is the same as the solution for line plots: turn axis auto-scaling off by setting the *[XYZ]Style* keywords to 1. The command you want to use is this.

```
IDL> Contour, data2d, lon, lat, XTitle=xtitle, $
        YTitle=ytitle, Title='Atmospheric Pressure', $
        XStyle=1, YStyle=1
```

Before we look at the new contour plot results, let's fix one other problem with the plot. Contour plots are not very useful to us unless the contour lines are labeled. And, usually, we need more contour lines than the default contour plot gives us. We can choose the number of contour lines we want to draw (sort of, I'll explain this in detail in just a moment) by setting the *NLevels* keyword. And we can choose which contour lines to label by setting the *C_Labels* keyword. The *C_Labels* keyword is a vector, whose value is set to 1 if you want that contour level labeled and to 0 if you do not want it labeled.

To draw, say, 12 contour levels and label them all, we can type this command. Note the use of the *Replicate* command, which replicates the value 1 twelve times and returns a 12-element vector.

```
IDL> Contour, data2d, lon, lat, XTitle=xtitle, $
        YTitle=ytitle, Title='Atmospheric Pressure', $
        XStyle=1, YStyle=1, NLevels=12, $
        C_Labels=Replicate(1, 12)
```

You see the result in Figure 3.

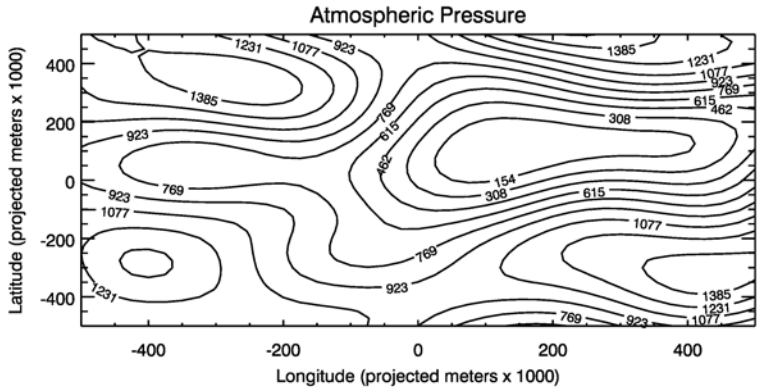


Figure 3: Finally, the contour plot is getting into the realm of the useful for us. Maybe the contour labels are too close together in certain areas of the contour plot. But this is a problem that can be addressed.

This contour plot is starting to contain the kind of information that will make it useful to us. But there are still modifications we can make to improve it even more. One problem we can fix, for example, is that the contour labels appear to be too close together in some areas of the plot, particularly in the upper right-hand corner. But we will address and solve this problem shortly.

Before we do, let's address another problem. The *Contour* command is rapidly becoming too long to type!

Advantages of Keywords versus System Variables

In addition to the 50 or so general purpose graphics keywords that apply to most IDL traditional graphics commands (you were introduced to perhaps 20 of these in the line plot chapter), there are perhaps 10-12 more

keywords that apply specifically to contour plots. Most of these additional keywords start with a "*C_*" prefix to identify them specifically as contour plot keywords. The *C_Labels* keyword is an example . These keywords can be discovered by consulting the IDL on-line help for the "CONTOUR procedure." IDL on-line help can be summoned like this.

```
IDL> ? contour
```

In IDL 8 and above, this will lead you straight to the new graphics IDL contour *function*, which is not what you need here. Locate the *Index* of the IDL on-line help application and find the *IDL Contour procedure*.

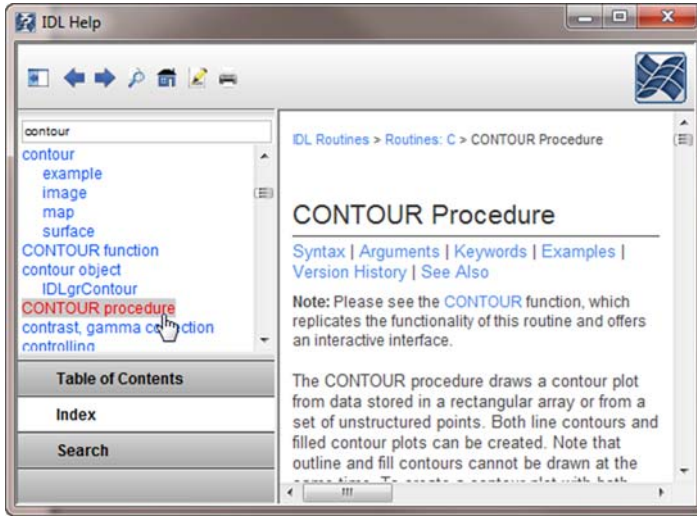


Figure 4: To learn which keywords are available for the IDL traditional graphics Contour command, be sure you locate the right entry in the IDL on-line help application. This shows the IDL help application for IDL 8.0.

Many of the standard graphics commands have counterparts in the IDL system variables, *!P*, *!X*, *!Y*, and *!Z*. I don't, in general, recommend setting these system variable counterparts because it is too easy to become confused about what is happening to your IDL graphics output when these system variables are set. Frankly, you will forget you have set them. Or, even if you remember you have set them, you will forget what you set them to.

In my own programming, I stay away from setting plotting system variables whenever possible. (And one of the reasons it takes me *hours* to

debug a problem with a colleague's IDL graphics program is that they *do* use graphics system variables in a program far removed from the one they have sent me to debug. Aaauugghhh!)

But, that said, I am going to break my rule. We are going to be using the same data, displayed much the same way, throughout this chapter. I don't want to type long contour commands any more than you do. And, yet, I want the examples in this book to be simple enough to type at the IDL command line.

So, here are the system variables I am going to use.

```
IDL> !X.Title = 'Longitude (projected meters x 1000)'
IDL> !Y.Title = 'Latitude (projected meters x 1000)'
IDL> !P.Title = 'Atmospheric Pressure'
IDL> !P.Color = cgColor('black')
IDL> !P.Background = cgColor('white')
IDL> !X.Range = [-500, 500]
IDL> !Y.Range = [-500, 500]
IDL> !X.Style = 1
IDL> !Y.Style = 1
```

Now, to produce the same plot as before, I only have to use the contour plot specific keywords. The command looks like this.

```
IDL> Contour, data2d, lon, lat, NLevels=12, $
       C_Labels=Replicate(1, 12)
```

Note: Remember, when you are typing an IDL keyword you only have to type enough letters to make it a unique keyword for the command. You don't have to type the full name of the keyword, although I always do for the sake of clarity. I wouldn't think of using a shortened keyword name if I was writing an IDL program. Short keyword names will make your programs especially hard for you and others to read and debug later. But, it's certainly okay to shorten keywords if you are noodling around at the IDL command line.

Customizing Contour Plots

Before we do anything else, let's fix the problem we identified earlier. The contour labels are too close together in some areas of the contour plot, particularly in the upper right-hand corner.

One way we can attempt to solve this problem is to make the contour labels themselves smaller. The contour labels are drawn by default with a

character size of 0.75. But we can set the size with the contour plot specific keyword *C_Charsize*. We could try, for example, a label size of 0.5.

```
IDL> Contour, data2d, lon, lat, NLevels=12, $
       C_Labels=Replicate(1, 12), C_Charsize=0.5
```

You see the result in Figure 5. This improves the situation somewhat, but almost makes the contour labels too small to read now.

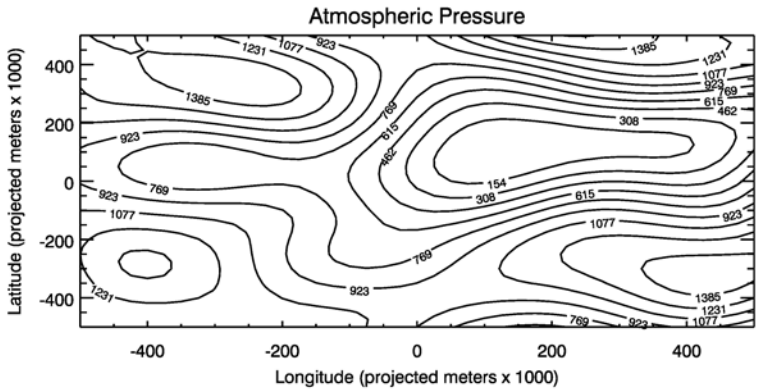


Figure 5: The contour labels can be sized independently of the contour plot titles and other annotations.

Unfortunately, it is not possible to determine exactly where the labels appear in traditional graphics contour plots. We do not have access to the algorithm that determines this property. But another solution we can try is to label every other contour interval. To do this, we need a vector of alternating 1s and 0s to pass to the *C_Labels* keyword. A quick way to create such a vector is like this.

```
IDL> everyOther = Reform(Rebin([1,0], 2, 6), 12)
IDL> Print, everyOther, Format='(12I2)'
       1  0  1  0  1  0  1  0  1  0  1  0
```

And we use it like this.

```
IDL> Contour, data2d, lon, lat, NLevels=12, $
       C_Labels=everyOther
```

Another way to label every other contour level takes us all the way back to IDL 4, but you still see it in use today. This method sets the *Follow* keyword. Originally, the *Follow* keyword would select the "contour-following," rather than the "cell-filling" algorithm for creating the contour

plot. One side-effect of the contour-following drawing algorithm was to label every other contour line. Today, all contour plots use the contour-following algorithm to draw the contours, but the keyword persists because of this useful side-effect. This command, then, has the same effect as the previous command.

```
IDL> Contour, data2d, lon, lat, NLevels=12, /Follow
```

You see the result in Figure 6.

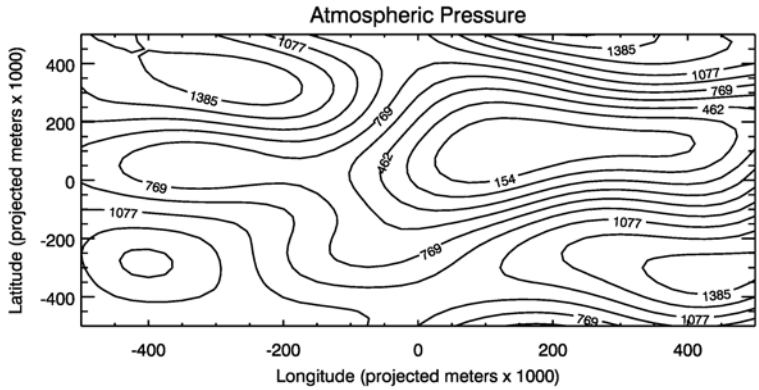


Figure 6: Labeling every other contour line can be accomplished with the modern C_Labels keyword or the much older Follow keyword.

Notice that neither of the contours in the lower-left corner of the plot are labeled, even though they are adjacent to one another. The smaller contour inside the larger should be labeled. But contour lines must have a non-specified minimum length to be labeled. This contour line is too short to be labeled. You don't have control over this, and the only solution is to make your contour plot larger, so that particular contour line will become long enough to be marked for labeling.

Selecting Contour Levels

The IDL documentation describes the *NLevels* keyword to the *Contour* command as representing the "number of equally spaced contour levels". If this keyword is not used, then "approximately six levels are drawn." This turns out to be a bit of fiction, as I will demonstrate in just a moment. In fact, the number of contour levels specified with the *NLevels* keyword is

highly dependent on the data being contoured and is rarely, if ever, equal to the number of levels you set with this keyword. It's true that as this number gets larger, more contour lines are drawn, but don't rely on the number of lines being accurate.

Consider the default case of "approximately six levels".

```
IDL> Contour, data2d, lon, lat, C_Labels=Replicate(1,6)
```

As you can see in Figure 7, the contour plot drew (maybe if we are being generous) three levels, at most. But nothing like six.

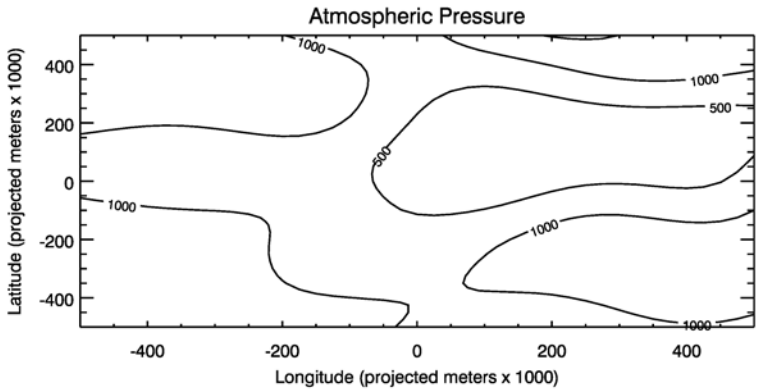


Figure 7: The NLevels keyword does not, generally, give you "N contouring levels" as claimed by the IDL documentation. Treat this as an approximation to the number of contour levels.

As it turns out with this particular data set, that when we ask for 12 contour levels, we actually get nine. But I will postpone this demonstration until the section dealing with filled contour plots, when this will become evident.

The bottom line is this. If you want N contour levels, then you need to define those levels yourself and specify them with the *Levels* keyword to the *Contour* command, rather than with the *NLevels* keyword. The *Levels* keyword is how you specifically select levels in your data to contour.

For example, if you want to draw contours at 250, 500, 750, and 1000 pressure units in this contour plot, you would type these commands.

```
IDL> levels = [250,500,750,1000]
IDL> Contour, data2d, lon, lat, Levels=levels, $
       C_Labels=Replicate(1,4)
```

You see the result in Figure 8.

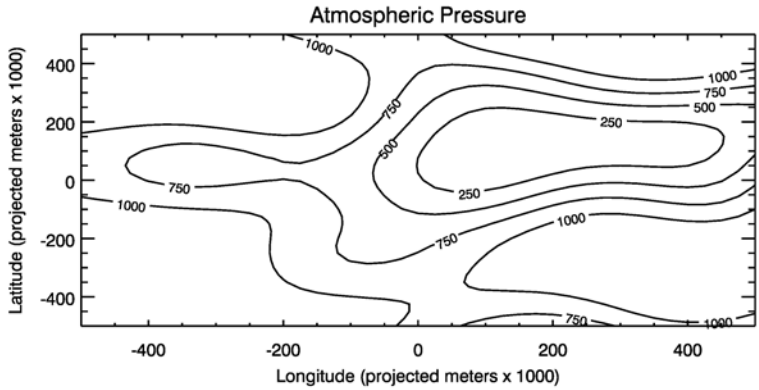


Figure 8: Contour levels can be selected and specified directly with the Levels keyword. For consistent results, the Levels keyword should be used instead of the NLevels keyword to specify contour levels.

If you want exactly 12 contour levels, you would have to calculate those levels yourself. Do not use the *NLevels* keyword. You would do it like this.

```
IDL> nlevels = 12
IDL> step = (Max(data2d) - Min(data2d)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(data2d)
IDL> Contour, data2d, lon, lat, Levels=levels, /Follow
```

You see the result in Figure 9. You can compare this with Figure 6 to see that this result is different from creating the contour plot as we did before with the keyword *NLevels=12*.

Note that the contour levels do not always have to be labeled with the value of the contour level. You can use the *C_Annotation* keyword to choose alphanumeric annotations for the contour levels. For example, you could label the contour levels 250, 750, and 1200 as “low,” “medium,” and “high,” like this.

```
IDL> threeLevels = [250, 750, 1200]
IDL> annotations = ['Low', 'Medium', 'High']
IDL> Contour, data2d, lon, lat, Levels=threeLevels, $
       C_Annotation=annotations
```

You see the result in Figure 10.

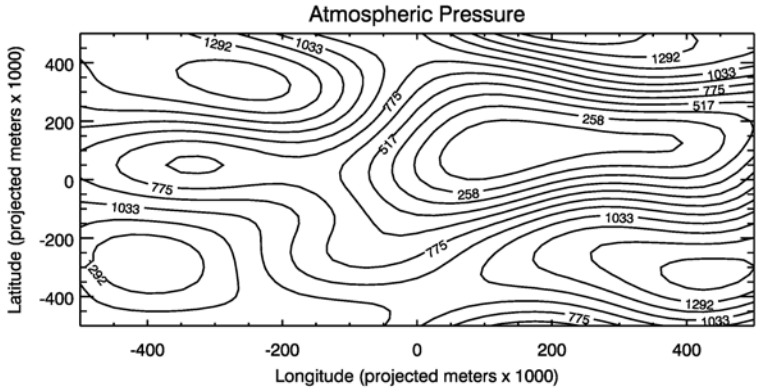


Figure 9: To get exactly 12 contour levels, you must calculate the levels yourself, rather than specifying 12 levels with the NLevels keyword.

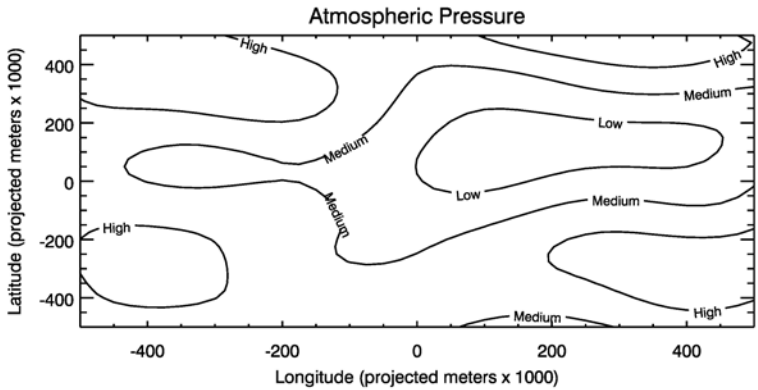


Figure 10: Contour levels can be labeled with alphanumeric labels.

Modifying Contour Lines

Contour lines can be drawn in the usual battery of line styles (see page 84). For example, to draw every other contour line in a dashed line style, we can use the *C_LineStyle* keyword like this.

```
IDL> Contour, data2d, lon, lat, Levels=levels, $
       C_LineStyle=[0,2], C_Labels=everyOther
```

You see the result in Figure 11.

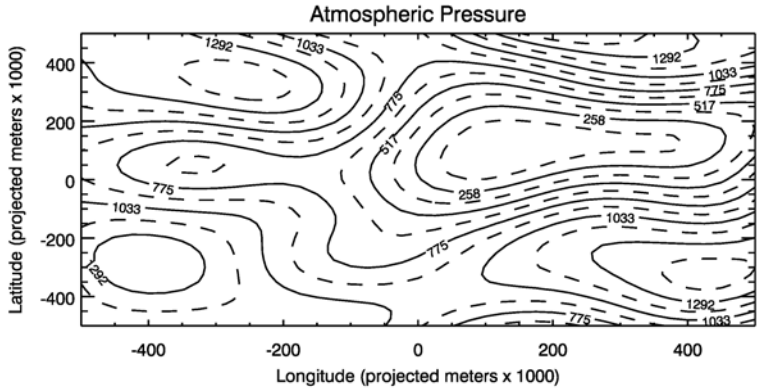


Figure 11: Contour lines can be displayed in different line styles. Use the C_Linestyle keyword to select a line style for each contour level.

Note that the *C_LineStyle* keyword is a "wrapping" keyword. In other words, its values "wrap around" and can be used over again if there are more than just two contour lines to draw. Most of the contour keywords are wrapping keywords. The one notable exception to this rule is the *C_Labels* keyword. If you set this keyword with just two values, as you just did for the *C_LineStyle* keyword, then only the first two contour lines will be labeled according to the values present in the vector. I don't know a good reason why the *C_Labels* keyword defies the rule. I've often had cause to wish it didn't.

Sometimes you want to make a thicker contour line at regular intervals. For example, we can make every third line a thicker line by using the *C_Thick* keyword like this.

```
IDL> everyThird = Reform(Rebin([0,0,1], 3, 4), 12)
IDL> Contour, data2d, lon, lat, Levels=levels, $
        C_Thick=[1,1,2], C_Labels=everyThird
```

You see the result in Figure 12.

It still might be difficult to determine which direction in a contour plot is the downhill direction. We can use the *Downhill* keyword to put small tick marks in the downhill direction of the contour lines, like this.

```
IDL> Contour, data2d, lon, lat, Levels=levels, $
        C_Thick=[1,1,2], C_Labels=everyThird, /Downhill
```

You see the result in Figure 13.

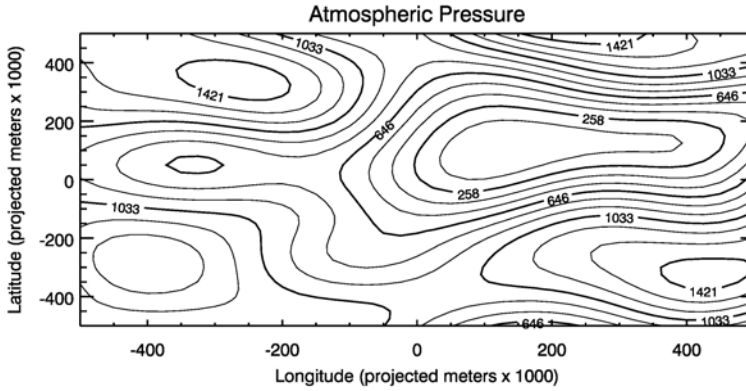


Figure 12: Every third line in the contour plot is made thicker with the C_Thick keyword.

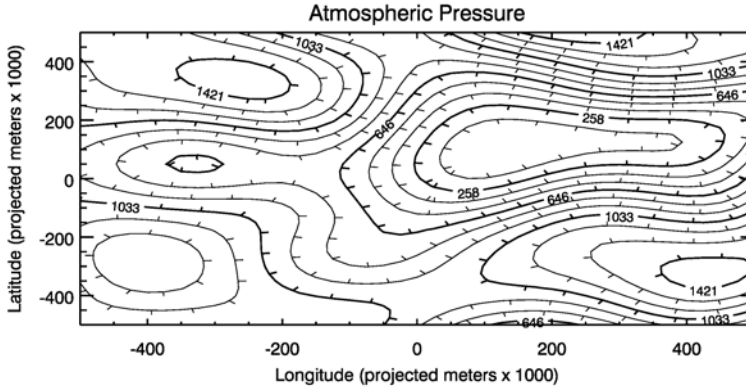


Figure 13: Ticks are placed in the downhill direction using the Downhill keyword.

Adding Color to Contour Plots

Contour plots can be drawn in a different color by setting the *Color* keyword. But, as with line plots, this will affect both the contour lines and the contour plot axes and annotations. Normally, when we are adding color to a contour plot, we want the axes to be one color and the contour lines to

be drawn in some other color or colors. The contour line colors can be selected with the *C_Colors* keyword to the *Contour* command.

The *C_Colors* keyword is a vector that describes, almost always, the color table indices with which each contour line is to be drawn. The only exception I am aware of is when these color indices are translated into 24-bit integers by cgColor so that the contour plot can be drawn using the color decomposition model. This, in fact, is how cgContour, a program you will learn more about later in this chapter, manages to draw color contour plots without ever loading colors into the physical color table.

This implies two very important points about adding color to contour plots. First, we need to load colors into the color table at the indices we specify in the *C_Colors* vector. And, then, we typically need to set ourselves up to be in indexed color mode to be able to see the proper colors. (See "Understanding IDL Color Models" on page 36 for additional information about color modes and models.) If we are using the default decomposed color mode, and we load color indices as the contour colors, then no matter what colors we have loaded into the color table, the contour colors will be displayed in shades of red!

I typically load contour colors at the bottom of the color table, so they don't interfere with the default locations of other drawing colors loaded with cgColor. I also often use Brewer color tables, rather than the ones supplied with IDL. If you do use IDL color tables, be careful which ones you select. Most of the IDL-supplied color tables use white and black as colors at either end of the color table. These are generally not the colors you want to use in a contour plot. If you use cgLoadCT to load IDL color tables, rather than *LoadCT*, you can use the *Clip* keyword to clip these troublesome colors from either end of the color table. (Black and white colors are sometimes used, however, to indicate either missing or totally saturated areas in the data.)

In the example here, there are 12 contour levels, so I need 12 contour colors. I choose to load these colors at the bottom of the color table, starting in color index 1. I do not ever load contour colors into color index 0 or 255, or it becomes impossible to produce PostScript output correctly. Since I depend on PostScript output to produce nice looking IDL plots for presentations, web output, and books, this is critical to me. Here is how I load colors from the IDL Blue-Red color table.

```
IDL> LoadCT, 33, NColors=12, Bottom=1
```

This color table has blue at one end of the color table and red at the other, so it is appropriate for contour colors. If I had wanted, say, to use the Standard Gamma II color table, I might have done something like this.

```
IDL> LoadCT, 5, NColors=12, Bottom=1
```

This color table has a black color at one end of the color table and a white color at the other end, as shown in Figure 14.

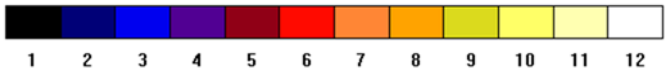


Figure 14: The twelve colors loaded by LoadCT for contour colors. Notice the black and white color at either end of the color range.

I can clip the black and white colors from this color table with cgLoadCT and the *Clip* keyword. In this example, colors are selected uniformly from the color table using the range of color indices 30 to 240, rather than from the usual 0 to 255.

```
IDL> cgLoadCT, 5, NColors=12, Bottom=1, Clip=[30,240]
```

You see the result in Figure 15.

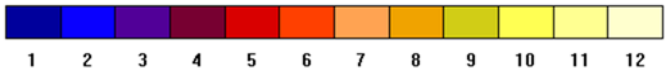


Figure 15: The same color table as before, but with the black and white colors "clipped" from each end by loading the colors with cgLoadCT and using the Clip keyword.

Using Colors in Contour Plots

Next, I set IDL to indexed color mode, saving the current color mode so I can set it back after I draw the contour plot.

```
IDL> Device, Get_Decomposed=currentMode
IDL> Device, Decomposed=0
```

Note: It is not possible to use both keywords at the same time with the Device command. If I do use both keywords, it will work correctly on Windows machines (the current mode will be fetched before it is changed to another mode), but it will fail on UNIX machines (the mode will be changed before the current mode is fetched). This operation must always be done in two steps to work in a device independent way. (Learn about the SetDecomposedState command below, which solves this problem for you.)

Finally, I draw the contour plot, using the *NoData* keyword to suppress drawing the contour lines. The contour lines are placed on the contour plot using the *Overplot* keyword and are drawn in color by setting the *C_Color* keyword to the proper color index values (in this case, 1 to 12). When I am done drawing the contour plot, I return to the color mode in effect before I changed the mode.

```
IDL> LoadCT, 33, NColors=12, Bottom=1
IDL> Contour, data2d, lon, lat, Levels=levels, /NoData, $
        Background=cgColor('white'), $
        Color=cgColor('black')
IDL> Contour, data2d, lon, lat, Levels=levels, $
        /Overplot, C_Colors=IndGen(12)+1, $
        C_Labels=everyOther
IDL> Device, Decomposed=currentMode
```

You see the result in Figure 16.

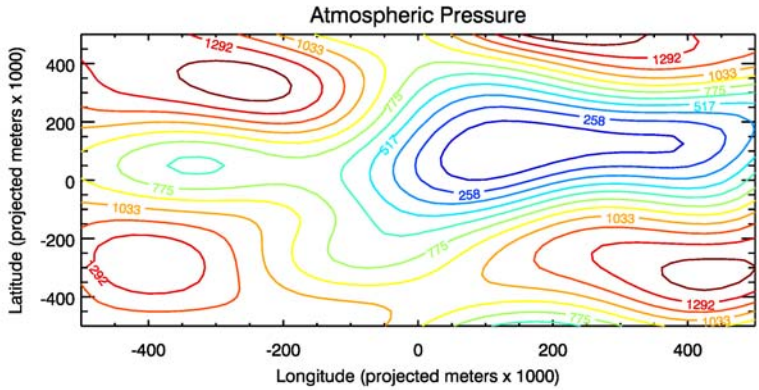


Figure 16: Contour lines can be drawn with colors, using the C_Colors keyword to indicate which color index in the color table should be used to select the color. This implies the indexed color mode is in effect when the plot is drawn.

Setting the Color Decomposition State

When we start to work with color table indices, such as those specified with the *C_Colors* keyword, in contour plots, we put ourselves in a position where we *must* use the indexed color model to display the colors. This is a problem with the code we have just written because it sets the

color decomposition state with the *Decomposed* keyword to the *Device* command.

Of course, this works perfectly on the display, because both normal display devices (i.e., X and WIN) accept a *Decomposed* keyword for the *Device* command. But some devices, such as older PostScript devices (older than IDL 7.1), do not. On those devices, the code above will throw an error on the first or second line of code!

The way we get around this problem is to use two programs from the Coyote Library that encapsulate the device and version dependencies so that we can obtain the current decomposition state, and set the decomposition state of the current graphics device in a device and version independent way. The two programs are GetDecomposedState to get the current color decomposition state or mode, and SetDecomposedState to set the current decomposition state or mode. The SetDecomposedState program uses GetDecomposedState to obtain the current color decomposition state before it is changed. This also solves the problem of having to get the decomposition state in one command for UNIX machines and set it with another (see "Setting the Color Model" on page 51 for more information). You will see these programs used in the code throughout the rest of this chapter.

Naturally, we can do other things with color and contour lines. For example, we could draw most lines blue, and every third line red, using code like this.

```
IDL> TVLCT, cgColor('blu4', /Triple), 1
IDL> TVLCT, cgColor('red4', /Triple), 2
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lon, lat, Levels=levels, /NoData, $
        Background=cgColor('white'), $
        Color=cgColor('black')
IDL> Contour, data2d, lon, lat, Levels=levels, $
        /Overplot, C_Colors=[1,1,2], C_Labels=everyThird
IDL> SetDecomposedState, currentState
```

You see the result in Figure 17.

Creating Color Filled Contour Plots

It is often when creating color filled contour plots that people run into the, well, peculiarities of the traditional graphics *Contour* command. It might be helpful to see how most people approach the problem of creating a

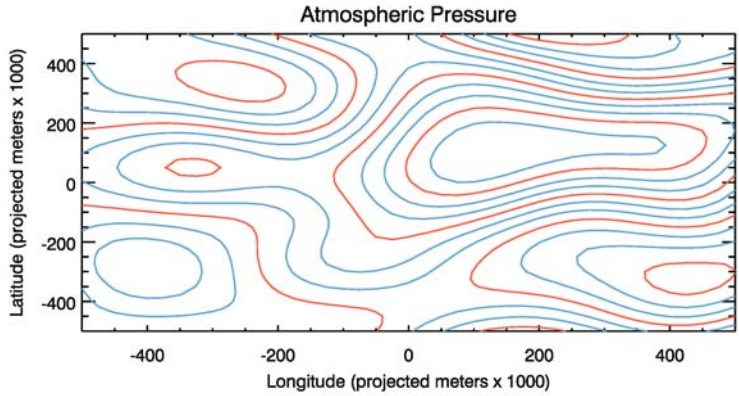


Figure 17: The C_Colors keyword is also a wrapping keyword. Here we draw contour lines in blue, except every third contour line is drawn in red.

color filled contour plot to see what some of the difficulties are and how to work around them.

Most people who want to create a color filled contour plot in IDL simply read the on-line documentation and see the directions call for loading a color table and setting the *Fill* keyword, and that's about it. Job done. Maybe they set *NLevels* to the number of contour levels they want to have in their filled contour plot.

Their first command might look something like this. Note I am adding a color bar to the contour command plot. This is common with filled color contour plots to give the user a sense of what the colors mean in the contour plot. IDL doesn't come with a traditional graphics color bar command, so I am going to use the cgColorbar program from the Coyote Library to produce the color bar above the contour plot. (Another alternative would be to use the discrete color bar command, cgDCBar, if I didn't have too many contour levels.) Notice how I save room for the color bar by using the *Position* keyword on the *Contour* command.

```
IDL> LoadCT, 33
IDL> Contour, data2d, lon, lat, /Fill, NLevels=12, $
        Position=[0.125, 0.20, 0.95, 0.75], $
        Background=cgColor('white'), $
        Color=cgColor('black'), XStyle=1, YStyle=1
IDL> Contour, data2d, lon, lat, /Overplot, $
        Color=cgColor('black'), NLevels=12, $
        C_Labels=everyOther
```

```
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $
       Divisions=12, XTicklen=1, XMinor=0, $
       AnnotateColor='black', Charsize=0.75, $
       Position=[0.125, 0.92, 0.95, 0.96]
```

What many people see on their display when they create this kind of color filled contour plot in IDL is shown in Figure 18.

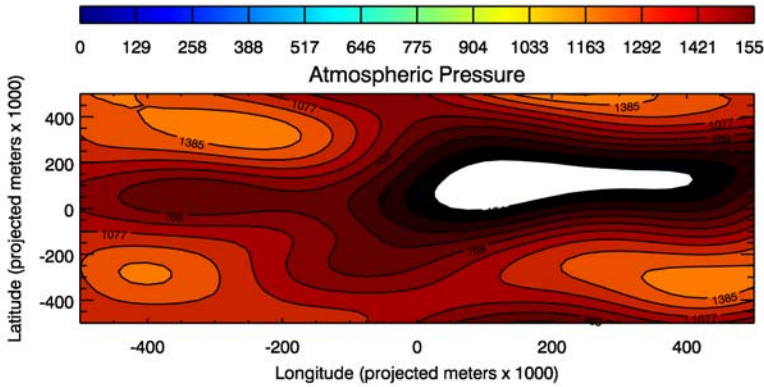


Figure 18: This is a typical result for many IDL users when they create a filled contour plot for the first time.

A number of things appear to be wrong with this contour plot. It certainly didn't use the colors we loaded into the color table. (You can see the contour plot is using different colors from those used in the color bar.) And there appears to be some kind of hole in the plot where we see the white background color. What in the world is going on here!?

Let's see if I can explain. The traditional graphics *Contour* command is very old. It was one of the first graphics commands added to IDL when the language was first written in the early 1980s. It was purchased from another software company to be added to IDL. Because it was purchased, the *Contour* command has always been something of a black box to IDL developers. Frankly, the code is not documented very well. It works, but no one seems to know exactly how it works!

Because of this, the *Contour* command has not changed much over the years. One way it hasn't changed is that in nearly 100 percent of the IDL programs I've ever seen, contour colors are expressed as indices into a color table. This means to use the *Contour* command successfully with

contour colors, you *must* be in indexed color mode. If you are not in indexed color mode, then you are by definition in decomposed color mode (the IDL default), and indexed colors in this mode will *always* appear in shades of red, no matter what colors you have loaded in the color table. The cgColorbar command is smart enough to put itself into the correct mode before it displays colors, but the *Contour* command is not.

So, you can solve the color problem and get yourself closer to what you want by just putting yourself in indexed color mode to use the *Contour* command.

```
IDL> LoadCT, 33
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lon, lat, /Fill, NLevels=12, $
        Position=[0.125, 0.125, 0.95, 0.80], $
        Background=cgColor('white'), $
        Color=cgColor('black'), XStyle=1, YStyle=1
IDL> Contour, data2d, lon, lat, /Overplot, $
        Color=cgColor('black'), NLevels=12, $
        C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $
        Divisions=12, XTicklen=1, XMinor=0, $
        AnnotateColor='black', Charsize=0.75, $
        Position=[0.125, 0.915, 0.955, 0.95]
```

You see the result in Figure 19.

We still have problems with this plot. For example, what about the colors themselves? We obviously are expecting 12 colors, since we asked for 12 contour levels with the *NLevels* keyword. In fact, (ignoring the white hole for a moment), we only find nine colors being used in the contour plot. On the other hand, the color bar looks like it is using 256 colors, but there appear to be some "extra" colors in the color bar itself. For example, there is a white line at about 388 and an extra black line about 517. Where are these colors coming from?

Two things are going on here. First, we have allowed IDL to choose colors from the color table for us. This is almost always a bad idea. By just setting the *Fill* keyword and not specifying which color indices we want to use with the *C_Colors* keyword we learned about in the previous section, IDL has "selected" some colors out of the color table. Which ones? Who knows! It would be better to load 12 colors at some defined location in the color table and use those specific 12 colors for both the contour plot and the color bar. We can modify the commands above to do this easily. We use the *NColors* and *Bottom* keywords when we load the colors, and we use

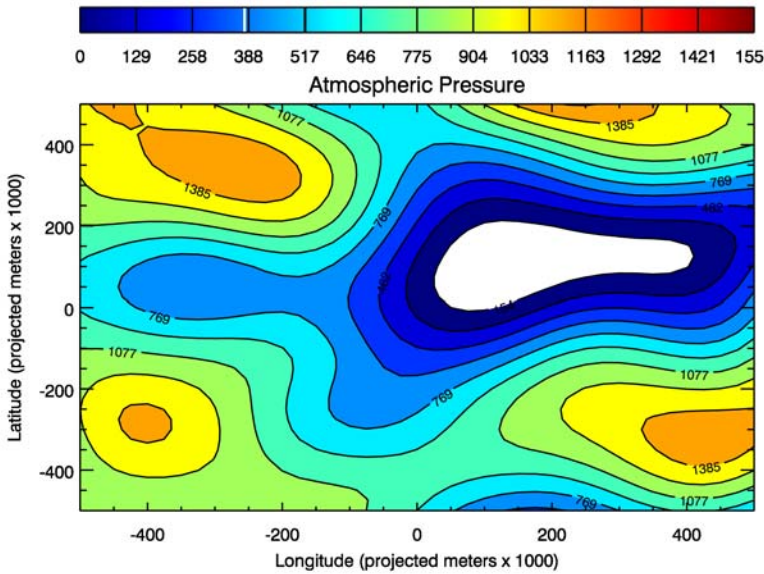


Figure 19: By running the Contour command in indexed color mode, the contour colors are (mostly!) correct.

C_Colors on the *Contour* command to specify those twelve color indices, starting at index 1 to be the colors we use in the contour plot.

```
IDL> LoadCT, 33, NColors=12, Bottom=1
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lon, lat, /Fill, NLevels=12, $
        Position=[0.125, 0.125, 0.95, 0.80], $
        Background=cgColor('white'), $
        Color=cgColor('black'), XStyle=1, YStyle=1, $
        C_Colors=IndGen(12)+1
IDL> Contour, data2d, lon, lat, /Overplot, $
        Color=cgColor('black'), NLevels=12, $
        C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $
        Divisions=12, XTicklen=1, XMinor=0, $
        AnnotateColor='black', NColors=12, Bottom=1, $
        Position=[0.125, 0.915, 0.955, 0.95], $
        Charsize=0.75
```

You see the result in Figure 20.

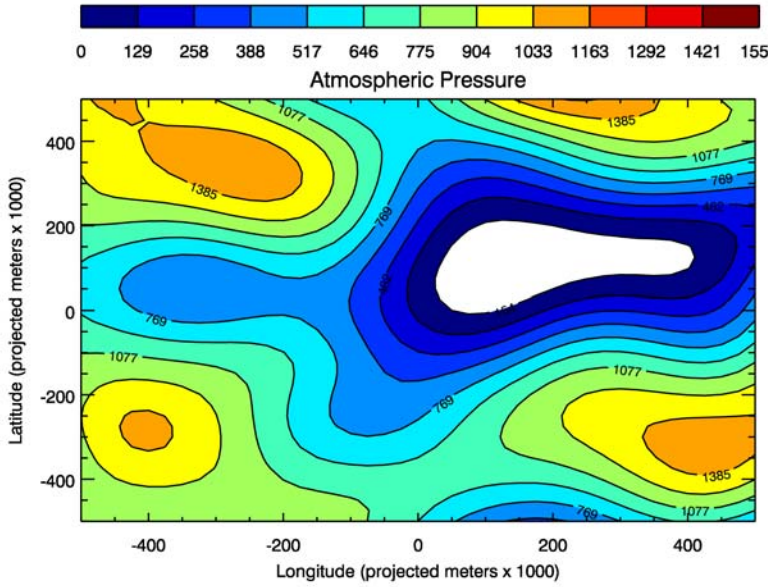


Figure 20: By choosing and loading only 12 colors, we can match the colors in the contour plot with the colors in the color bar. Well, almost. We still have a small problem that we are not using 12 colors in the contour plot!

If you look carefully, you will notice that the problem of extra lines in the color bar has gone away in this version of the contour plot. How come?

What was happening is that when you use indexed color mode your color table can become "dirty" or "contaminated" from other programs using it. There is only one color table and all programs running in IDL in indexed color mode get their colors from that same color table. If a program is forced to load colors in the color table, and you use that table without first loading it with the colors you expect to be there, those colors could well be wrong.

In our case, since we set ourselves in indexed color mode, and then used cgColor to load drawing colors for the contour plot, cgColor was forced to use the color table. This corrupted the color table and those corrupted colors later showed up when we used all 256 colors for the color bar.

There is an absolute rule about using indexed color mode that you want to memorize. When using indexed color mode, always, *always* load the col-

ors you want to use from the color table *immediately* prior to using them. Don't assume they will be correct!

By restricting the colors to the bottom of the color table as we are doing in the last command, we don't run into, or use, colors that may have been loaded by cgColor. We are corrupting the color table for sure, but it is not doing us any harm, since we have separated the drawing colors from the contour fill colors. (If you want to see the colors you have loaded in the current color table, type CIndex.)

Note: I should mention that it is not absolutely required that indexed colors be used in contour plots. It is possible to use decomposed colors. In fact, cgContour, which you will learn about later in this chapter (on page 162), uses decomposed colors whenever possible. It uses cgColor to convert color table index values to decomposed color values. (See "Using cgColor With Color Table Indices" on page 45.) But I have seen thousands of IDL programs with contour plots in them, and cgContour is the only one I have ever seen that works with decomposed colors. That's why I think it is important to know how to set up indexed contour colors correctly.

With this plot it is now obvious we are only using nine contour intervals, not 12. For example, there are no red colors in the plot, even though we have red colors in the 12 colors of the color bar. It is also obvious that the labels on the contour lines don't match the values on the color bar. Why not?

The reason is that we have let IDL choose the contour intervals for us with the *NLevels* keyword, and whatever algorithm IDL is using to do this is not giving us the kind of results we expect. To create 12 contour intervals *exactly*, and have the color bar values match the contour labeling, we must create the contour levels ourselves. Then, we must use the *Levels* keyword to specify the levels and forget we ever heard about the *NLevels* keyword.

The code becomes something like this.

```
IDL> nlevels = 12
IDL> LoadCT, 33, NColors=nlevels, Bottom=1
IDL> step = (Max(data2d) - Min(data2d)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(data2d)
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lon, lat, /Fill, Levels=levels, $
        Position=[0.125, 0.125, 0.95, 0.80], $
        Background=cgColor('white'), $
        Color=cgColor('black'), XStyle=1, YStyle=1, $
        C_Colors=IndGen(nlevels)+1
```

```
IDL> Contour, data2d, lon, lat, /Overplot, $
        Color=cgColor('black'), Levels=levels, $
        C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $
        Divisions=12, XTicklen=1, XMinor=0, $
        AnnotateColor='black', NColors=12, Bottom=1, $
        Position=[0.125, 0.915, 0.955, 0.95], $
        Charsize=0.75
```

You see the result in Figure 21.

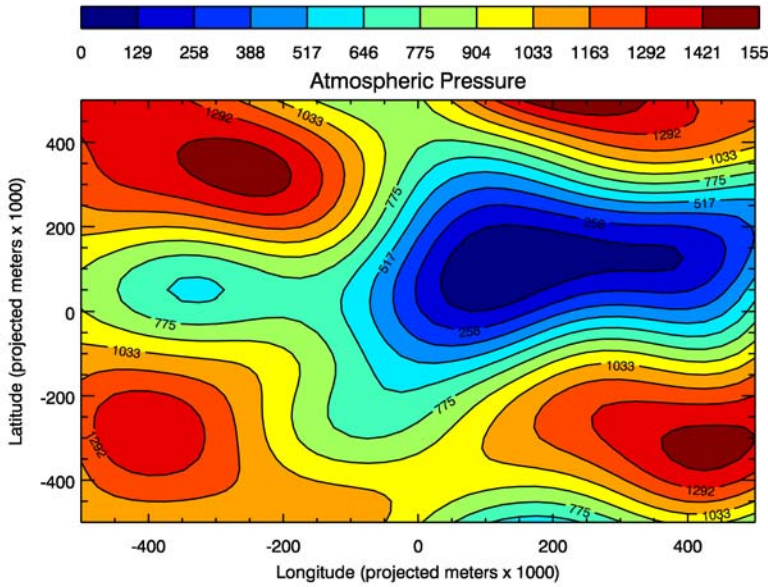


Figure 21: By specifying your own contour levels, the contour labels match the values in the color bar, and the hole in the contour plot disappears.

Notice that one consequence of defining your own contour levels is that the hole in the contour plot disappears! The other major benefit, of course, is that now your contour labels agree with the values in the color bar, and all 12 contour levels are colored with the appropriate colors.

Choosing a Different Fill Algorithm

There are occasions when the *Fill* keyword is not appropriate for creating filled contour plots. The algorithm that the *Contour* command uses to fill

contours can become confused if there are "open" contours to fill. Open contours are sometimes created by missing data and are often created by drawing filled contours on map projections set up with the *Map_Set* command. If you are having trouble with your filled contour plots, always try replacing the *Fill* keyword with the *Cell_Fill* keyword.

The *Cell_Fill* keyword uses a "cell filling" algorithm that is slightly less efficient and slower than the algorithm used by setting the *Fill* keyword. The upside, however, is that it is sometimes more accurate because it draws more contour polygons to fill. In the case of the data we have been using, the results would be identical for both keywords.

Filled Contours on Map Projections

If you are placing filled contour plots on map projections (especially those created with the *Map_Set* command), an excellent rule of thumb is to *always* use the *Cell_Fill* keyword rather than the *Fill* keyword. It is notoriously easy to inadvertently create open contours with map projections, and the danger is that if you don't use *Cell_Fill*, the colors of your contours may be incorrect. And, what is worse, it will be extremely difficult to tell this is the case.

The only trick in putting filled contour plots on map projections is that all three positional parameters are required parameters in this case, and you must overplot the filled contours onto the map by setting the *Overplot* keyword.

Suppose we use this same data set, but adjust the X and Y positional parameters so they represent locations in the United States, represented by a longitude range of -124° to -66° and a latitude range of 25° to 50°. Recall that the data variable, *data2d*, is a 41 x 41 array. Normally, when I have to create variables of a particular size, with explicit endpoints, I use the Coyote Library routine Scale_Vector to do so. We can create the proper X and Y vectors like this.

```
IDL> lats = Scale_Vector(Findgen(41), 25, 50)
IDL> lons = Scale_Vector(Findgen(41), -124, -66)
```

Now we are ready to set up the map projection data coordinate space using the *Map_Set* command, like this. Note that we have to erase the window with a background color with a map projection to get the background color we want. The *NoErase* keyword on the *Map_Set* command is essential here to prevent the background color from being erased.

```
IDL> Erase, Color=cgColor('white')
IDL> Map_Set, /Mercator, 37.5, -95, /NoBorder, $
        Limit=[25, -124, 50, -66], $
```

```
        Color=cgColor('black'), $
        Position=[0.05, 0.05, 0.95, 0.80], /NoErase
```

We put the filled contour command on the map like this. Here I am using a Brewer color table, whose colors are designed to work well on maps. I load the Brewer color table with the Coyote Library routine cgLoadCT.

```
IDL> nlevels = 12
IDL> step = (Max(data2d) - Min(data2d)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(data2d)
IDL> cgLoadCT, 4, NColors=nlevels, Bottom=1, /Brewer, $
       /Reverse
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lons, lats, /Cell_Fill, $
       Levels=levels, C_Colors=IndGen(nlevels)+1, $
       /Overplot
IDL> Contour, data2d, lons, lats, /Overplot, $
       Color=cgColor('grey'), Levels=levels, $
       C_Labels=everyOther
IDL> SetDecomposedState, currentState
```

Finally, we can add the map outlines and the color bar to complete the plot.

```
IDL> Map_Continents, Color=cgColor('charcoal')
IDL> Map_Continents, Color=cgColor('charcoal'), /USA
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $
       Divisions=nlevels, XTicklen=1, XMinor=0, $
       AnnotateColor='black', NColors=nlevels, $
       Bottom=1, Position=[0.1, 0.87, 0.9, 0.90], $
       Title='Atmospheric Pressure', Charsize=0.75
```

You see the result in Figure 22.

Contouring Irregularly Sampled Data

The data we have passed to the *Contour* command so far has always been gridded two-dimensional data. Sometimes our data are not like that. For example, if we are collecting atmospheric pressure data, as we have been supposing in the examples so far, we may be sending weather balloons up to collect the data. Such data will be distributed in random locations over the sampling area.

Such a data array *must* be gridded before it can be contoured, but we have a couple of ways to proceed. We can either grid the data array ourselves and then pass it to the *Contour* command. Or, we can simply ask the *Con-*

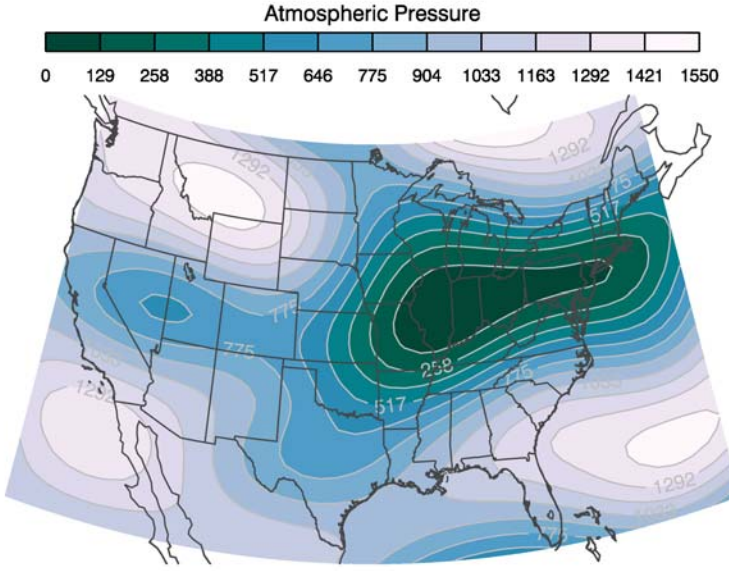


Figure 22: Be sure to use the Cell_Fill keyword to add filled contour plots to map projections created with the Map_Set command Map projections often create open contours that the Fill keyword doesn't handle well.

tour command to grid the data array for us, before contouring it, by setting the *Irregular* keyword. We have more control, and more options, if we grid the data array ourselves, so I will show you a simple gridding method you can use with your data.

It might be interesting to sample the data set we are using now, just to see how well we can reproduce the results with irregularly sampled data. To do so, we need to make two-dimensional arrays of our X and Y vectors. We can do that like this.

```
IDL> lat2d = Rebin(Reform(lat, 1, 41), 41, 41)
IDL> lon2d = Rebin(lon, 41, 41)
```

Let's create 200 random points from this data set, add some random noise to the data, so they don't fall directly on a grid, and display them in a plot to see where they are located.

```
IDL> pts = Round(RandomU(-3L, 200) * 41 * 41)
IDL> dataIrr = data2d[pts]
```

```
IDL> lonIrr = lon2d[pts] + RandomU(5L, 200) * 50 - 25
IDL> latIrr = lat2d[pts] + RandomU(8L, 200) * 50 - 25
IDL> Plot, lonIrr, latIrr, PSYM=4, XRange=[-500, 500], $
        YRange=[-500,500], XStyle=1, YStyle=1, $
        Title='Random Sampling Locations'
```

Note: In the code above I used particular numbers for the random number seed (i.e., -3L, 5L, and 8L). This is so your results will be identical to the results shown here. If you want truly random values, use the variable "seed" as the seed in the RandomU commands above.

You see the result in Figure 23.

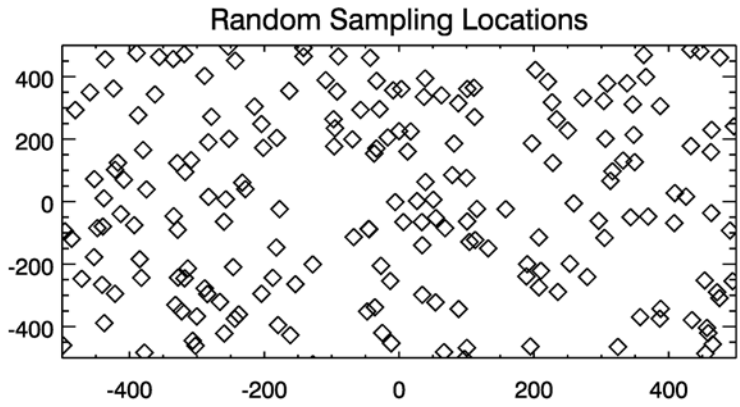


Figure 23: Here are the simulated random sampling locations of the data values we want to contour.

The simplest way to grid irregular data yourself is to use the two IDL routines *Triangulate* and *Trigrid*. This is the method used by the *Contour* command itself, although you don't have access to the parameters of the routines when doing the gridding directly with the *Contour* command.

The *Triangulate* command is used to produce a set of Delaunay triangles that the *Trigrid* command can use to produce the gridded data. Delaunay triangulation is an algorithm for producing a set of triangles from a set of points, such that a circle formed by connecting the vertices of any triangle does not contain any other point. Triangulations are sometimes plagued by collinear points. (One of the reasons contouring irregular data directly with the *Contour* command will sometimes fail.) You can use the *Repeat*

and *Tolerance* keywords to the *Triangulate* command to deal with circumstances like this.

To create the set of Delaunay triangles and return them in an output *triangles* variable, we use the *Triangulate* command.

```
IDL> Triangulate, lonIrr, latIrr, triangles
```

Now we are ready to pass these triangles to the *Trigrid* command. We can set the size of the output grid we want with the *NX* and *NY* keywords. The default grid size is 51 by 51. Since we are trying to reproduce the original data, we will set these keywords to produce a grid of 41 by 41.

The important thing in gridding irregular data to contour is to return from *Trigrid* the X and Y vectors that specify the locations of the gridded data. These can be obtained from the *XGrid* and *YGrid* output keywords. The command will look like this.

```
IDL> gridData = Trigrid(lonIrr, latIrr, dataIrr, $
        triangles, NX=41, NY=41, $
        XGrid=xgrid, YGrid=ygrid)
```

We can display this gridded data as a filled contour plot.

```
IDL> nlevels = 12
IDL> step = (Max(gridData) - Min(gridData)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(gridData)
IDL> LoadCT, 33, NColors=nlevels, Bottom=1
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, gridData, xgrid, ygrid, /Cell_Fill, $
        Levels=levels, Background=cgColor('white'), $
        Position=[0.125, 0.125, 0.95, 0.80], $
        Color=cgColor('black'), XStyle=1, YStyle=1, $
        C_Colors=IndGen(nlevels)+1
IDL> Contour, gridData, xgrid, ygrid, /Overplot, $
        Color=cgColor('black'), Levels=levels, $
        C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(gridData),Max(gridData)], $
        Divisions=12, XTicklen=1, XMinor=0, $
        AnnotateColor='black', NColors=12, Bottom=1, $
        Position=[0.125, 0.915, 0.955, 0.95], $
        Charsize=0.75
```

You see the result in Figure 24. Compare this result with Figure 21. Not too bad, considering we used less than 15 percent of the original data points to construct the gridded data set.

This result should not look too different from doing the gridding directly with the *Contour* command by setting the *Irregular* keyword.

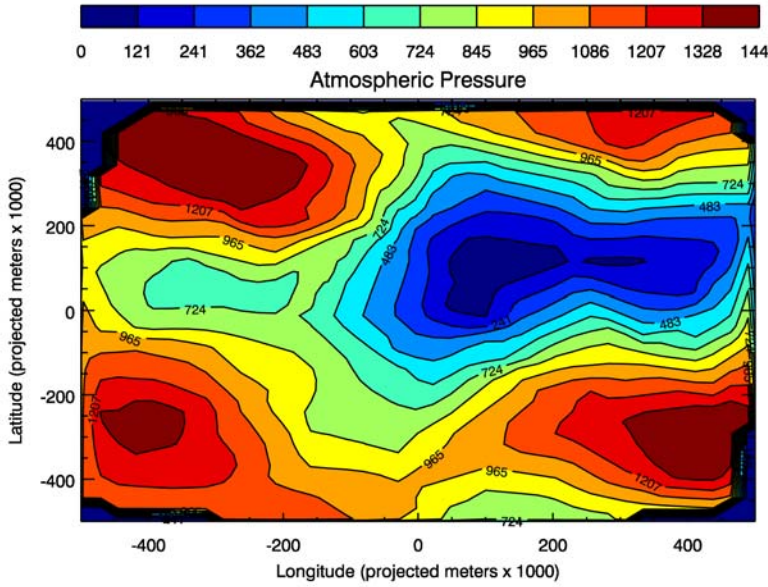


Figure 24: Irregular data can be gridded with Triangulate and Trigrid before being contoured.

```
IDL> nlevels = 12
IDL> step = (Max(dataIrr) - Min(dataIrr)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(dataIrr)
IDL> LoadCT, 33, NColors=nlevels, Bottom=1
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, dataIrr, lonIrr, latIrr, /Fill, $
        Levels=levels, Background=cgColor('white'), $
        Position=[0.125, 0.125, 0.95, 0.80], $
        Color=cgColor('black'), XStyle=1, YStyle=1, $
        C_Colors=IndGen(nlevels)+1, /Irregular
IDL> Contour, dataIrr, lonIrr, latIrr, /Overplot, $
        Color=cgColor('black'), Levels=levels, $
        C_Labels=everyOther, /Irregular
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(gridData),Max(gridData)], $
        Divisions=12, XTicklen=1, XMinor=0, $
        AnnotateColor='black', NColors=12, Bottom=1, $
        Position=[0.125, 0.915, 0.955, 0.95], $
        Charsize=0.75
```

You see the result in Figure 25 .

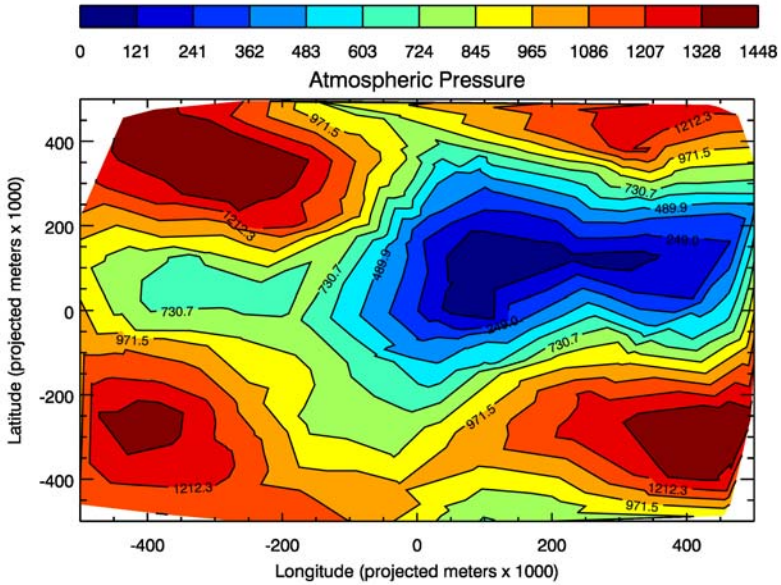


Figure 25: The same irregular data gridded by the contour command directly by setting the Irregular keyword.

Comparing Figure 25 to Figure 24, you see one obvious difference. The blue border around the outside of Figure 24 is missing in Figure 25. What is the *Contour* command doing that we didn't do when we gridded the data ourselves?

It turns out that the *Contour* command sets all gridded data that falls outside the Delaunay triangle boundary (or, another way of say this is outside the "convex hull" of the data points) to the missing value NaN (not a number). We can do this ourselves with the *Missing* keyword to *TriGrid*. Because we didn't use the *Missing* keyword, all our missing data values got set to 0. We probably want to set our missing values to something other than 0, but this introduces several other complications I think you should know about.

Suppose we decide to set all data outside the convex hull to the missing value NaN. Floating point NaNs are represented in IDL with the system variable *!Values.F_NAN*. We would set the missing values like this.

```
IDL> gridData = Trigrid(lonIrr, latIrr, dataIrr, $
       triangles, NX=41, NY=41, $
       XGrid=xgrid, YGrid=ygrid, Missing=!Values.F_NAN)
```

But, in the rest of our code, we have to be careful to handle missing data correctly. For example, we have these two lines in our display code.

```
IDL> step = (Max(gridData) - Min(gridData)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(gridData)
```

Look what happens when we check the values of these variables.

```
IDL> Help, step
     STEP    FLOAT   =  NaN
IDL> Print, levels
     NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN
```

Yikes! Those kinds of values are not going to create a very good looking contour plot. To create the correct values, we are going to have to be careful to set the *NaN* keyword on these functions to properly exclude NaN values.

```
IDL> step = (Max(gridData, /NaN) - Min(gridData, /NaN)) $
       / nlevels
IDL> Help, step
     STEP    FLOAT   =  118.689
IDL> levels = IndGen(nlevels) * step + $
       Min(gridData, /NaN)
IDL> Print, levels
     23.9834      142.672      261.360      380.049
     498.737      617.426      736.115      854.803
     973.492      1092.18      1210.87      1329.56
```

We have several of these functions in our display code. Here is the modified code.

```
IDL> nlevels = 12
IDL> step = (Max(gridData, /NaN) - Min(gridData, /NaN)) $
       / nlevels
IDL> levels = IndGen(nlevels) * step + $
       Min(gridData, /NaN)
IDL> LoadCT, 33, NColors=nlevels, Bottom=1
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, gridData, xgrid, ygrid, /Fill, $
       Levels=levels, Background=cgColor('white'), $
       Position=[0.125,0.125,0.95,0.80], $
       Color=cgColor('black'), XStyle=1, YStyle=1, $
       C_Colors=IndGen(nlevels)+1
IDL> Contour, gridData, xgrid, ygrid, /Overplot, $
```

```
        Color=cgColor('black'), Levels=levels, $
        C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(gridData, /Nan), $
        Max(gridData, /NaN)], Charsize=0.75, $
        Divisions=12, XTicklen=1, XMinor=0, $
        AnnotateColor='black', NColors=12, Bottom=1, $
        Position=[0.125, 0.915, 0.955, 0.95]
```

You see the result in Figure 26.

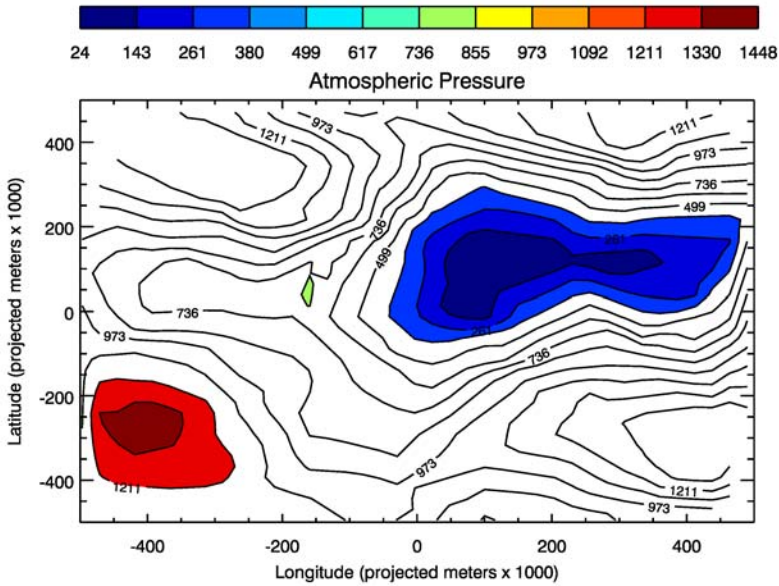


Figure 26: Something bad has happened to our filled contour plot!

Something bad is going on here! What has happened is that by introducing NaNs into the data, we have caused "open contours" to appear in the contouring algorithm. The *Fill* keyword to the *Contour* command selects an algorithm that doesn't know how to cope with open contours very well. To correct the problem, we have to choose the *Cell_Fill* keyword instead.

Making just this one change in the code above, produces the result you see in Figure 27.

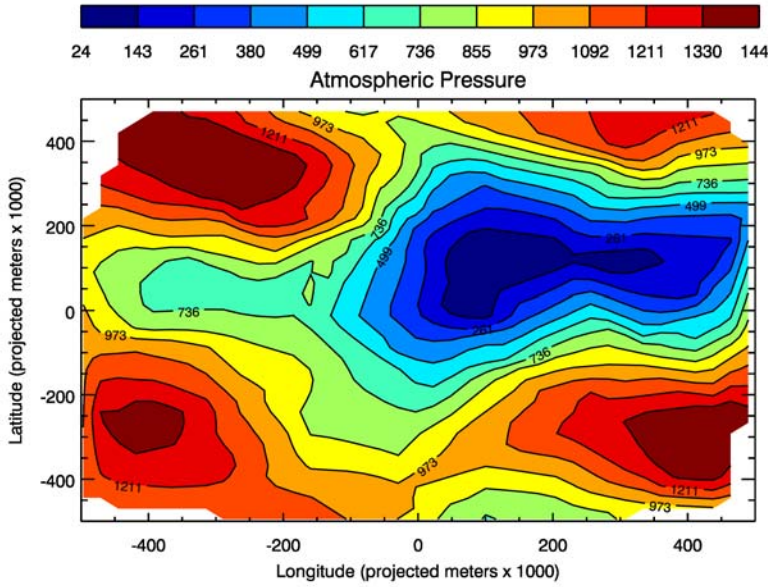


Figure 27: By changing the Fill keyword to Cell_Fill, we produce the correctly filled contour plot.

This result can now be compared favorably to the *Contour* result itself in Figure 25.

Gridding Irregular Data

An even more powerful method of gridding data, because it has many more options that the *Triangulate/Trigrid* method we just discussed, is to use the IDL routine *GridData* in combination with the *QHull* command which can build Delaunay triangles. Let's try to use it to build a similar grid for contouring.

First, we build the set of Delaunay triangles, using *QHull*.

```
IDL> QHull, lonIrr, latIrr, triangles, /Delaunay
```

Next, we grid the data, using the triangles we just created. We have our choice of several different gridding methods, among which are inverse distance, kriging, linear interpolation, nearest neighbor, polynomial fitting and so on. We are going to use the inverse distance method, in which data points closer to the grid are weighted more heavily than grid points further away.

```
IDL> gridData = GridData(lonIrr, latIrr, dataIrr, $
        Method='InverseDistance', Dimension=[41,41], $
        Start=[-500, -500], Delta=[25, 25], $
        Triangles=triangles, Missing=!Values.F_NaN)
```

So that we can use our current *lon* and *lat* vectors, we set the *Start*, *Delta*, and *Dimension* keywords to produce a grid that is identical to our original gridded data set.

We display the data like this.

```
IDL> nlevels = 12
IDL> step = (Max(gridData, /NaN) - Min(gridData, /NaN)) $
        / nlevels
IDL> levels = IndGen(nlevels) * step + $
        Min(gridData, /NaN)
IDL> LoadCT, 33, NColors=nlevels, Bottom=1
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, gridData, lon, lat, /Fill, $
        Levels=levels, Background=cgColor('white'), $
        Position=[0.125,0.125,0.95,0.80], $
        Color=cgColor('black'), XStyle=1, YStyle=1, $
        C_Colors=IndGen(nlevels)+1
IDL> Contour, gridData, lon, lat, /Overplot, $
        Color=cgColor('black'), Levels=levels, $
        C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(gridData, /Nan), $
        Max(gridData, /NaN)], Charsize=0.75, $
        Divisions=12, XTicklen=1, XMinor=0, $
        AnnotateColor='black', NColors=12, Bottom=1, $
        Position=[0.125, 0.915, 0.955, 0.95]
```

You see the result in Figure 28. You have to be somewhat careful in how you interpret contouring results. Look, for example, in the lower right corner of this figure. Are there really two peaks in the data? Possibly, but we can't be sure. It is not that the contouring algorithm is wrong or that the result is incorrect; it is that the resolution of the data does not always lend itself to unambiguous results. It is always a good idea to filter the result of any data analysis through your own head before you tell the world about your amazing discovery.

Contouring a Real World Example

Naturally, contour plots always work great when you are learning about them in a book. But real world examples are more difficult. They tend to

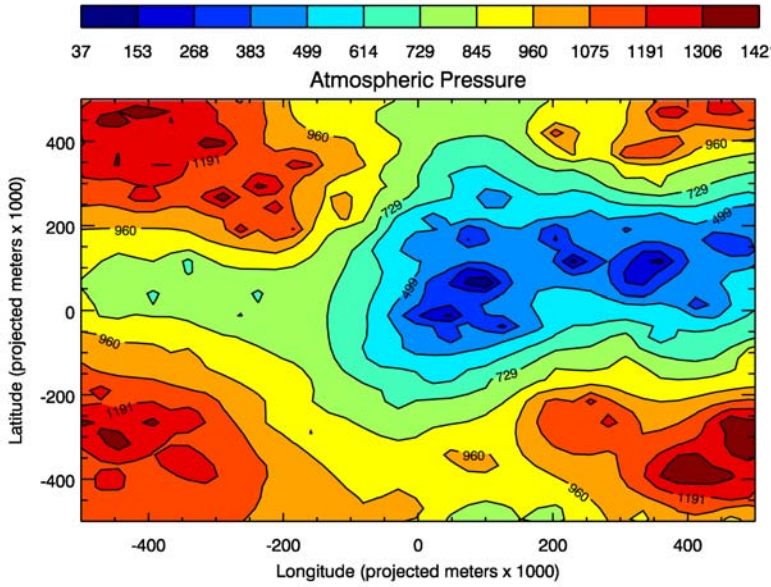


Figure 28: The same irregular data gridded with GridData using an inverse weighting gridding method.

throw curve balls, or—as my cricket playing Australian colleagues say—"wrong'uns." They are like projects around the house. You can plan on three trips to the hardware store, no matter how simple you think the job is going to be.

So, here is a real world example that gives you some practice applying the techniques you have learned so far in this chapter. The data set we will use is a publicly available NASA Goddard Satellite-based Surface Turbulent Fluxes (GSSTF) data set (*http://disc.sci.gsfc.nasa.gov/measures/documentation/Readme.GSSTF2b.pdf*). I am going to show you how to contour the latent heat flux from a data set from 1 January 2008, which is stored in an HDF-EOS5 scientific data format (an HDF5 format). The file is named *GSSTF.2b.2008.01.01.he5*. If you like, you can download the file from my web page, although this is not necessary. The data file is quite large and I have saved the required variables from the file in an IDL save file, which I will describe shortly, and which is much faster to download. Here is the URL to obtain the data file itself, if you care to use it.

http://www.idlcoyote.com/books/tg/data/GSSTF.2b.2008.01.01.he5

The latent heat flux variable in the file is named "E" and the fill value is named "_FillValue." Case is important in scientific data format files. Here is the IDL code I used to read the data, the fill value, and the dimensions of the data from the file. It is necessary to reset the IDL session before we start to clear away any system variable values we were using earlier in the chapter. (This is not necessary if you are starting your IDL session with this example. Nor is it necessary to type these commands. You can skip this command block and find an easier way to obtain the relevant data in the paragraph below the command block.)

```
IDL> .RESET
IDL> file = 'GSSTF.2b.2008.01.01.he5'
IDL> fileID = H5F_Open(file)
IDL> dataName = '/HDFEOS/GRIDS/SET1/Data Fields/E'
IDL> dataID = H5D_OPEN(fileID, dataName)
IDL> dataspaceID = H5D_GET_SPACE(dataID)
IDL> dims = H5S_GET_SIMPLE_EXTENT_DIMS(dataspaceID)
IDL> lon_dim = dims[0]
IDL> lat_dim = dims[1]
IDL> latentHeat = H5D_READ(dataID)
IDL> fill_valueID = H5A_OPEN_NAME(dataID, '_FillValue')
IDL> fill_value = H5A_READ(fill_valueID)
IDL> H5A_Close, fill_valueID
IDL> H5D_Close, dataID
```

If you chose not to download the data file and type the commands listed above, you can follow along with this example by restoring the file *heat_flux.sav* from among the data sets you downloaded to use in this book. The file can be found here if you haven't yet downloaded it.

http://www.idlcoyote.com/books/tg/data/heat_flux.sav

The variables *latentHeat*, *fill_value*, *lon_dim*, and *lat_dim* will be restored.

```
IDL> Restore, File='heat_flux.sav'
IDL> Help, latentHeat, fill_value, lon_dim, lat_dim
     LATENTHEAT      FLOAT     = Array[360, 180]
     FILL_VALUE      FLOAT     = Array[1]
     LON_DIM         ULONG64   =  360
     LAT_DIM         ULONG64   =  180
```

The first curve ball you see here is that the *fill_value* variable is a one-element array, instead of the scalar value you were probably expecting. This is going to set up a classic "gotcha" situation with the *Where* function in

IDL, which is what we must use to find these "missing" or fill values in the data if we plan to exclude them from further processing.

Look what happens if you are not aware that the *fill_value* variable is a one-element array.

```
IDL> indices = Where(latentHeat EQ fill_value, count)
IDL> Print, count
      1
```

Compare this result to using a scalar value for the *fill_value* variable.

```
IDL> indices = Where(latentHeat EQ fill_value[0], count)
IDL> Print, count
      38774
```

Quite a difference! While you can get away with using a one-element array for a scalar most of the time in IDL, here is one example where you definitely cannot. When IDL evaluates the expression with the EQ operator and two vectors, it quietly truncates the result to match the smaller of the two vectors. The *Where* function is the unfortunate victim in this operation. Yikes!

So, the first thing to do is to take care of this scalar versus array problem by making the fill value a scalar value.

```
IDL> fill_value = fill_value[0]
```

The fill value essentially identifies the "missing" data values in this file. In order to calculate the proper levels for contouring, we need to remove these missing values from consideration. The normal way to do this is to set these missing values to NaNs. The code looks like this.

```
IDL> indices = Where(latentHeat EQ fill_value, cnt)
IDL> IF cnt GT 0 THEN latentHeat[indices] = !Values.F_NAN
```

Note that we didn't have to convert the *latentHeat* variable to a floating variable first, because it already is a floating point array.

Now we can find the minimum and maximum of the remaining "good" data. Remember to set the *NaN* keyword.

```
IDL> minData = Min(latentHeat, MAX=maxData, /NaN)
IDL> Print, minData, maxData
      -13.5637       583.412
```

Normally, latent heat flux is shown in about 8 regular divisions of about 50 watts per square meter. We could easily create 8 contour levels for this data set, but if we start from the minimum value of the data, as we have done previously in this chapter, the levels will not be "natural" divisions.

They will have arbitrary floating point values such as -13.5637, 63.5637, 133.5637, etc. It would be better to have 8 levels that started at 0 and went up in units of 50 (0, 50, 100, etc.), but then we have the problem that some of the data in the file will have values less than the minimum level or greater than the maximum level.

We could solve this problem by adding two additional "levels" or contour colors, one for values less than 0 and one for values greater than the highest contour level we are interested in. In other words, we divide the data into 10 colors or divisions, with the lowest and highest division representing the values that are outside the range of values we are particularly interested in. We can create 10 divisions by specifying the nine levels for the contour plot like this. (The top division, the 10th, represents all the values greater than the value of our last contour level.)

```
IDL> levels = [minData, IndGen(8)*50]
```

Next, this data is world-wide data on a one-degree grid, so we can set up the longitude and latitude vectors we need to display the data with like this.

```
IDL> lon = Scale_Vector(Findgen(lon_dim),-180,179) + 0.5
IDL> lat = Scale_Vector(Findgen(lat_dim), -90, 89) + 0.5
```

The extra 0.5 we add to each element is to align the left (longitude) or bottom (latitude) edge of the grid cell with a whole degree. The centers of the grid cells then fall on half-degree increments (-179.5°, -178.5°, -177.5°, etc.). The advantage of this kind of gridding is that it avoids the non-physical extra half degree "below" the South Pole (at -90.5°) and the small gap just below the North Pole (at 89.5°).

We are going to display this in a window that is wider than it is tall, so we need a *Window* command. But, we want this program to work in the PostScript device, too, so we need to protect this *Window* command and only issue the command if we are on a device that supports windows.

```
IDL> IF (!D.Flags AND 256) NE 0 THEN $
        Window, XSize=1000, YSize=700
```

Another way to do this is to use the cgDisplay command, which automatically protects the command in a PostScript device. In fact, in a PostScript device, it creates a "window" having an aspect ratio of 700/1000, just like this window on the display device.

```
IDL> cgDisplay, 1000, 700
```

I would like to display the filled contour plot in a window with a white background. Since I want to put this on top of a map projection, and map projections don't allow me to use a *Background* keyword to set the back-

ground color, I have to erase the window with the background color, and then use a *NoErase* keyword on my map projection command. I set the map projection (a cylindrical projection, in this case) up like this.

```
IDL> Erase, Color=cgColor('white')
IDL> Map_Set, Position=[0.05,0.05,0.95,0.75], /NoErase
```

Next, we load the 16 colors for the contour plot, and display the filled contour plot. We will use a Brewer color table, since Brewer colors were designed to work well on maps. The code looks like this.

```
IDL> cgLoadCT, 25, /Brewer, NColors=10, Bottom=1, /
        Reverse
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, latentHeat, lon, lat, /Overplot, $
        /Cell_Fill, C_Colors=IndGen(10)+1, $
        Levels=levels, Color=cgColor('black')
IDL> Contour, latentHeat, lon, lat, /Overplot, $
        Levels=levels, Color=cgColor('black')
IDL> SetDecomposedState, currentState
```

Note: We are using the Cell_Fill keyword rather than the Fill keyword in this example. This is essential on maps, generally, which tend to create open contours, and even more so when there are open contours as a result of missing values (NaN) in the data being contoured.

The next step is to add map annotations. I would like to display this with box style axes, and I want the character size in a PostScript file to be slightly smaller than the character size I use on the display. I set the character size like this.

```
IDL> charsize = (!D.Name EQ 'PS') ? 0.65 : 1.0
IDL> Map_Continents, Color=cgColor('tan'), /Fill
IDL> Map_Continents, Color=cgColor('black')
IDL> Map_Grid, /Box_Axes, Color=cgColor('black'), $
        Charsize=charsize
```

Finally, we add a color bar to the plot to indicate the data values and their associated colors. In this case, we use the discrete color bar routine cgDCBar from the Coyote Library.

```
IDL> labels = ['< 0','0-50', '50-100', '100-150', $
        '150-200', '200-250', '250-300', '300-350', $
        '350-400', '>400']
IDL> cgDCBar, NColors=10, Bottom=1, Color='black', $
        Position=[0.05, 0.9, 0.95, 0.94], Rotate=-45, $
        Labels=labels, Charsize=charsize, $
        Title='Latent Heat (watts/meter^2)'
```

You see the result in Figure 29.

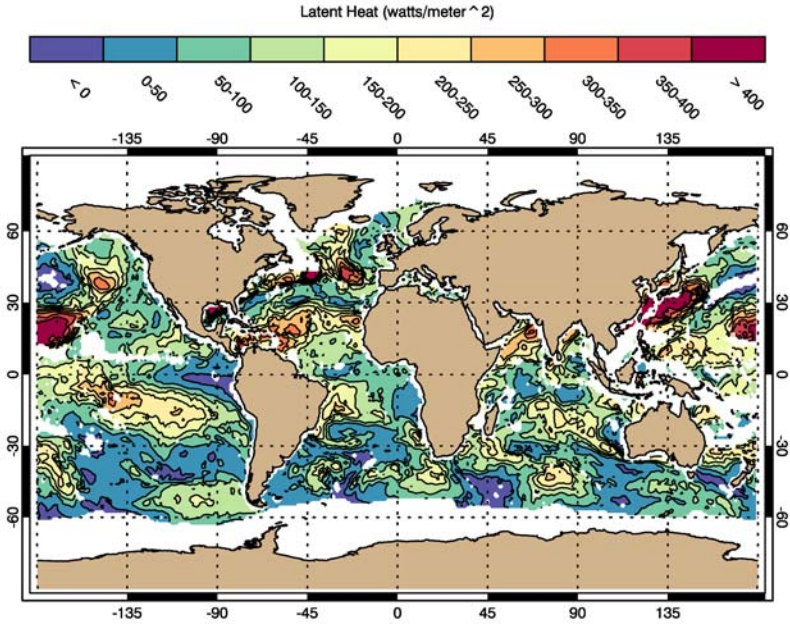


Figure 29: A real world example of contouring latent heat flux from a NASA satellite data set.

Using a Refurbished Contour Command

You will find that many of the problems identified with the traditional *Contour* command in IDL have been corrected in the cgContour command from the Coyote Library. The cgContour command is part of a suite of programs, collectively called the Coyote Graphics System (which includes cgPlot, cgPlotS, cgSurf, cgText, and cgWindow), for creating traditional graphics output that work and look the same on all devices and in any color decomposition state. The cgContour command takes care of many of the details necessary to write device independent graphics programs, works with colors in a more natural way by specifying colors directly, and has features that are not available in the *Contour* command. It can easily be displayed in the resizeable graphics window, cgWindow.

The Coyote Graphics commands produce, by default, black output on white backgrounds. This is the opposite of IDL traditional commands, but makes it possible to use these commands to produce (as much as possible) identical looking PostScript output. (You can return to the traditional

color scheme of white on black by setting the *Traditional* keyword on the command.)

You will notice that textual output from these commands is slightly larger than normal. There are two reasons for this. First, larger output more closely matches the look and feel of the corresponding PostScript output when PostScript or TrueType fonts are used. And, second, larger output more closely matches the look and feel of similar object graphics programs in IDL 8.

The cgContour program has been specifically written to address the following problems with the *Contour* command:

- The *NLevels* keyword should specify exactly *N* contour levels.
- There should be no "hole" in a filled contour plot.
- There should be an easy selection method for which contour levels to label.
- It should draw graphics output in decomposed color mode, when possible, so color tables do not become "polluted" with drawing colors.

Here are some side-by-side comparisons to give you a sense of how cgContour works. Notice that, by default, all contour levels are labeled. Here is an example of the default output of *Contour* and cgContour. (If you have been working the examples in this chapter, and you haven't already done so, you might want to start a fresh IDL session by entering the *.Reset* executive command at the IDL command prompt before you begin with these commands.)

```
IDL> data = cgDemoData(2)
IDL> LoadCT, 0, /Silent
IDL> Window, 0, XSize=400, YSize=400
IDL> Contour, data, Title='Normal Contour Plot', $
        XTitle='X Title', YTitle='Y Title'
IDL> Window, 1, XSize=400, YSize=400
IDL> cgContour, data, Title='Coyote Contour Plot', $
        XTitle='X Title', YTitle='Y Title'
```

You see the result in Figure 30.

Here is a similar comparison with basic filled contour plots. First, set up the common elements.

```
IDL> cgLoadCT, 17, /Brewer
IDL> cgLoadCT,4, NColors=10, Bottom=1, /Brewer, /Reverse
```

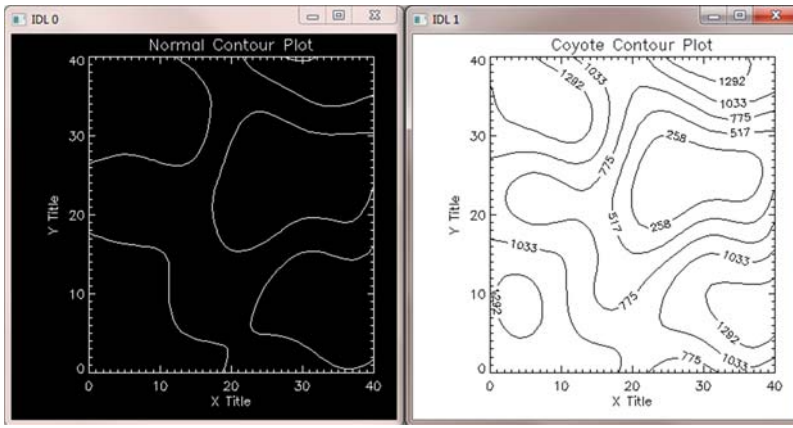


Figure 30: A side-by-side comparison of the basic Contour command versus the basic cgContour command.

```
IDL> c_colors = IndGen(10) + 1
IDL> position = [0.1, 0.1, 0.9, 0.8]
```

Now, draw the contour plot with the *Contour* command. Make sure you preserve the color model and that you draw the filled contours using the indexed color model.

```
IDL> Window, 0, XSize=400, YSize=400
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data, NLevels=10, /Fill, $
       Position=position, C_Colors=c_colors
IDL> Contour, data, NLevels=10, /Overplot
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Divisions=10, NColors=10, Bottom=1, $
       Range=[Min(data),Max(data)], TickLen=1.0, $
       Position=[0.1,0.90,0.90,0.94], Charsize=0.75
```

Now, draw the same filled contour plot with cgContour.

```
IDL> Window, 1, XSize=400, YSize=400
IDL> cgContour, data, NLevels=10, /Fill, $
       Position=position, C_Colors=c_colors
IDL> cgContour, data, NLevels=10, /Overplot
IDL> cgColorbar, Divisions=10, NColors=10, Bottom=1, $
       Range=[Min(data),Max(data)], TickLen=1.0, $
       Position=[0.1,0.90,0.90,0.94], Charsize=0.75
```

You see the result in Figure 31. Notice you don't have to worry about setting the color decomposition state, since the cgContour program takes care of this automatically.

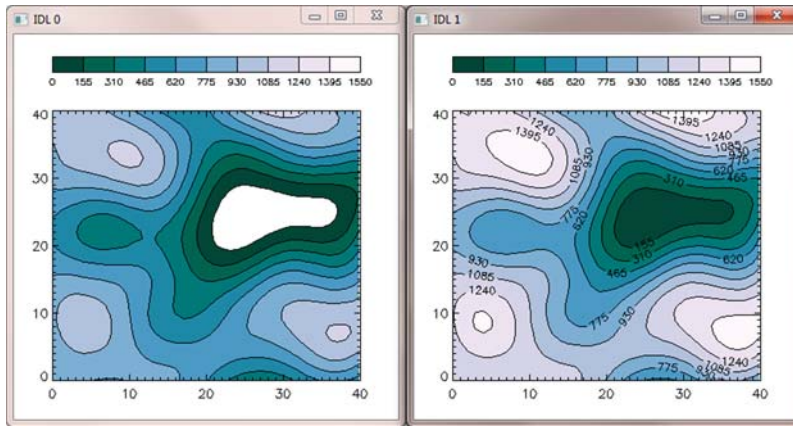


Figure 31: A side-by-side comparison of a filled contour plot with the Contour command and a filled contour plot with the cgContour command.

The cgContour command also has the ability to accept a color palette so that contour colors are completely independent of the colors loaded in the current color table. The easiest way to obtain a color palette is with the cgLoadCT command and its *RGB_Table* output keyword. If this keyword is used, colors are not loaded into the current color table, but are simply loaded into a color palette and returned to the user. The plot on the right in Figure 31 can be reproduced with a Standard Gamma II color table like this.

```
IDL> cgLoadCT, 5, RGB_Table=pal
IDL> Window, 2, XSize=400, YSize=400
IDL> cgContour, data, NLevels=10, /Fill, $
        Position=position, Palette=pal
IDL> cgContour, data, NLevels=10, /Overplot
IDL> cgColorbar, Divisions=10, NColors=10, Bottom=1, $
        Range=[Min(data),Max(data)], TickLen=1.0, $
        Position=[0.1,0.90,0.90,0.94], Charsize=0.75, $
        Palette=pal
```

You see the result in Figure 32.

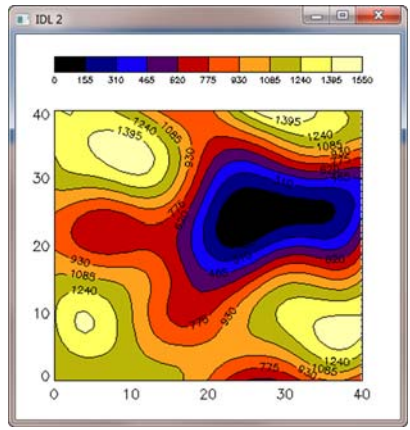


Figure 32: The cgContour command allows you to specify a color table palette that keeps your contour plot colors completely independent of the colors in the current color table.

Labeling Contour Intervals

The cgContour command also makes it easy to choose which contours to label. If you want every other contour, or every third contour, labeled, you simply set the new *Label* keyword to 2 or 3. Set it to 0 to turn contour labeling completely off, like the default for the *Contour* command. The default for cgContour is 1, which labels every contour level.

Using Colors in Contour Plots

The cgContour command makes it very easy to use colors in contour plots. The axis and annotation color can be different from the contour colors. And, these colors can be expressed as color names that are among the 200 color names recognized by cgColor. Consider these commands that show how colors can be used in a different way than with the *Contour* command.

```
IDL> Window, 0, XSize=400, YSize=400
IDL> cgContour, data, NLevels=10, AxisColor='blue', $
       Color='red'
IDL> Window, 1, XSize=400, YSize=400
IDL> cgContour, data, NLevels=5, AxisColor='brown', $
       C_Colors=['aquamarine', 'dark green', 'orange', $
       'crimson', 'purple']
```

You see the result in Figure 33.

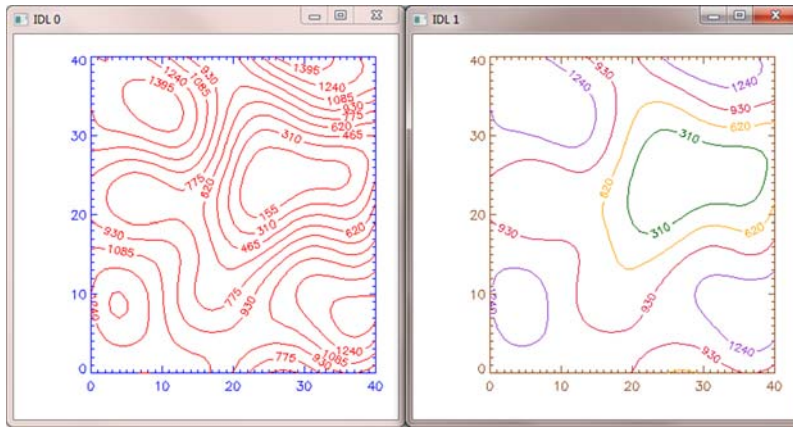


Figure 33: The cgContour command allows color names to be used to specify contour colors.

Contour Plots in Resizeable Graphics Windows

If you want to see the contour plot by itself in a resizeable graphics window, with controls for making hardcopy output files, set the *Window* keyword. The contour plot is displayed in cgWindow.

```
cgContour, data, NLevels=10, AxisColor='navy', $
   Color='red', /Window
```

You see the result in Figure 34.

The cgWindow application allows you to write the graphics commands to a PostScript file, or save the display in any of five different raster file formats. If you have ImageMagick installed on your machine, you have the option of creating raster files by converting PostScript files to raster output with ImageMagick's *convert* command. This results in raster files of significantly higher quality, especially in the quality of the fonts used for annotation. You will learn more about this topic in "Presentation Quality by Leveraging PostScript" on page 405.

You can add as many graphics commands as you like to a cgWindow program window. Here, for example, is how to display a filled contour plot with contour lines overlaid, and with a color bar at the top of the plot. Simply set the *Window* and/or *AddCmd* keywords on the commands to display the contour plot in a resizeable graphics window.

```
IDL> data = cgDemoData(2)
IDL> cgLoadCT, 4, /Brewer, /Reverse, NColors=12, $
        Bottom=1
```

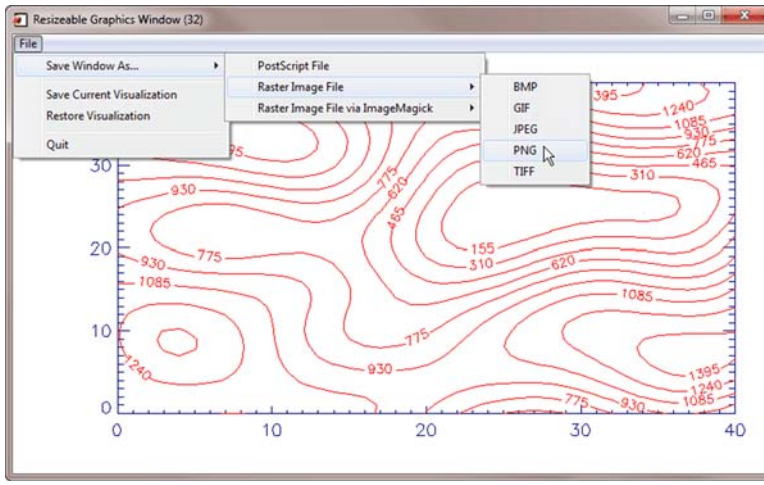


Figure 34: The cgContour command can be displayed in a resizeable graphics window, where you can save the graphics window in five different raster file formats and as a PostScript file. This resizeable graphics window is created with the Coyote Library program cgWindow.

```
IDL> cgContour, data, NLevels=12, $
        /Fill, C_Colors=IndGen(12)+1, $
        Position=[0.1,0.1,0.9,0.75], /Window
IDL> cgContour, data, NLevels=12, $
        Color='Charcoal', /Overplot, /AddCmd
IDL> cgColorbar, Divisions=12, $
        Range=[Min(data), Max(data)], NColors=12, $
        Bottom=1, XMinor=0, XTicklen=1.0, /AddCmd
```

You see the result in Figure 35.

If you would like to list the commands you have loaded into the cgWindow command list, just set the *ListCmd* keyword. The commands are printed in the console window.

```
IDL> cgWindow, /ListCmd
     0. cgContour, p1, C_COLORS=value, FILL=value, $
          NLEVELS=value, POSITION=value
     1. cgContour, p1, COLOR=value, NLEVELS=value, $
          OVERPLOT=value
     2. cgColorbar, BOTTOM=value, DIVISIONS=value, $
          NCOLORS=value, RANGE=value, XMINOR=value, $
          XTICKLEN=value
```

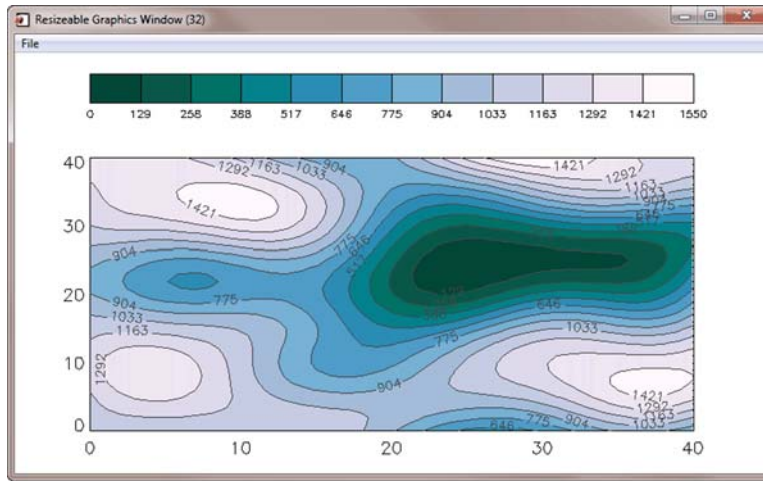


Figure 35: The cgWindow program allows you to add an unlimited number of graphics commands to the graphics window. The graphics window is resizeable, and the window content can be sent to a PostScript file or saved in any of five different raster file formats.

You can use the resizeable graphics window cgWindow just like you use other graphics windows in IDL. Use cgSet to select a cgWindow to add commands to, and cgDelete to delete cgWindow programs you are finished with. You can have as many cgWindow programs on the display as you like, each with a different display, as in normal IDL graphics windows.

For example, here is how you can save the window index number of the first cgWindow, open a second cgWindow, and display another contour plot.

```
IDL> windowIndex = cgQuery(/Current)
IDL> cgWindow
IDL> cgContour, cgDemoData(18), NLevels=20, Label=2, $
        Color='charcoal', /AddCmd
```

The cgQuery function is used to obtain information about the cgWindow programs currently on the display. You can determine their window index numbers, the widget identifiers of their top-level base widgets, the window titles, or obtain their object references by setting appropriate keywords. If the keyword *Current* is set, as it was here, this information is returned for just the current cgWindow (i.e., the last one created).

To display another contour plot in the first cgWindow, type this.

```
IDL> cgSet, windowIndex
IDL> cgContour, cgDemoData(18), NLevels=20, Label=2, $
        Color='dodger blue', /Window
```

You see the result in Figure 36.

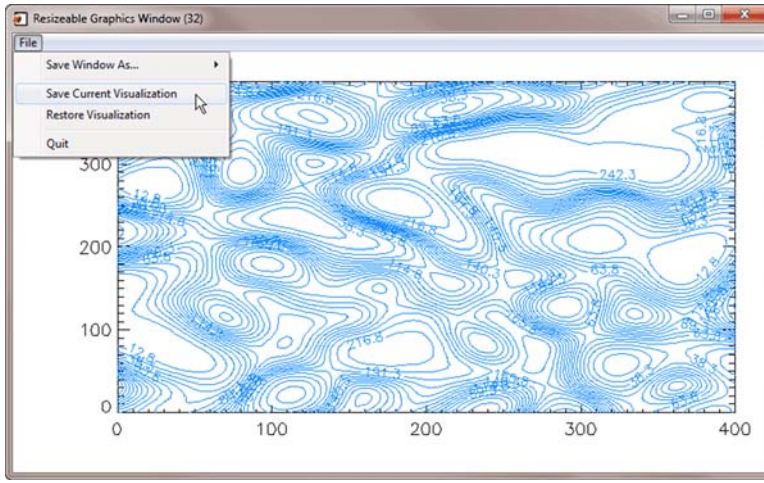


Figure 36: You can use resizeable graphics windows in exactly the same way you currently use normal graphics windows. Here a new command was added to the window.

The graphics display (the *visualization*) in a cgWindow can be saved to a file, where it can be e-mailed to a colleague so he or she can view the same visualization you see, or so you can restore it later to view it again. Simply choose the Save Current Visualization button from the pull-down File menu, as illustrated in Figure 36.

To delete the first cgWindow, type this.

```
IDL> cgDelete, windowIndex
```

The second cgWindow will be sent forward on your display so you can view it easily.

You can delete all the cgWindow applications currently on the display by setting the *All* keyword.

```
IDL> cgDelete, /All
```