IDIDL—IMAGE DISPLAY AND MANIPULATION; AND COLORBARS
August 29, 2011

Carl Heiles

Contents

1. PIXELS, COLORS, BYTES

Your display screen consists of about a million little areas called *pixels*. Each pixel can show a different color/intensity combination. Everything on your screen—text, pictures, whatever—is displayed by filling the appropriate pixels with the appropriate color/intensity. Your screen dimensions might be 1280×1024 (5×4 aspect ratio)—found on many computers of all kinds (known as SXGA); the somewhat larger 1440×900 (8×5 aspect ratio), known as SXGA+; or 1024×768 (4×3 aspect ratio)—found on older computers and small laptops. These aspect ratios come in much larger dimensions, like 1600×1200 pixels for 4×3. Each pixel is small! But if you look carefully, you can see them.

Typical CCD projectors (as used for PowerPoint) have the 4×3 format with 1024×768 pixels as their native resolution, so you get best results, with no image cropping, when you set your video card's resolution to this value.

2. DECOMPOSED COLOR (i.e., TrueColor) *vs.* COMBINED COLOR (i.e., PseudoColor)

All colors seen by the human eye can be produced by a suitable mixture of intensities of only three colors: red, green, and blue (RGB). Most common displays in use today allow 256 intensities of each color[1]. This gives a total of 256^3 combinations—this used to be billed in the PC world as "millions of colors". This is called *TrueColor*.

Believe it or not, it is often desireable to degrade the full "millions of colors" true scheme to a 256-color scheme; this is called *PseudoColor*. In Pseudocolor, you are *combining* the separate RGB colors into a single 256-element set; that's why it's called *Combined color*. The particular set of 256 colors chosen is called the *colortable*; each of the 256 colors is a particular combination of RGB.

The most common colortable is the grayscale one. In this, the intensities of RGB are all identical. They change uniformly from zero intensity (for entry number 0—this makes black) to full intensity (for entry number 255—this makes white). In between, we have all shades of gray. An image made with the grayscale colortable looks like a black-and-white photograph. If you choose a non-gray colortable, then you have what's often called "false color", which can be helpful in highlighting certain features or achieving more contrast. We give an example below, and other examples in our "1d2d3d: One, Two, and Three Dimensional Color Images" memo.

2.1. Combined Color: 256-Element Colortables

In your computer's memory, an image consists of a two-dimensional array of *data* values d, one for each pixel. These are actual numbers. Their range is restricted: d can take on 256 different values, ranging from 0 to 255. On your screen, the image consists of projected light in each pixel, which we call the intensity I.

There is a one-to-one relationship between d and I, so there are also 256 different possible values for I. Generally speaking, this relationship is specified by the "color table". The color table is often nonlinear so as to emphasize weak or strong features. Sometimes it's an equal mixture of red, green, and blue (grayscale); sometimes the mixture is engineered to produce color. By "Intensity" (I), we mean the 256 different possible combinations of light intensity and color, one for each value of d.

To use combined color in IDL, you must turn the native 256^3 RGB colors into a 256-element colortable, which you accomplish by turning color decomposition off. You do this with the command

```
device, decomposed=0
```

By default, IDL sets the three colortables equal, which gives you grayscale. You can redefine this;

[1]Note that 256 is the same as 2^8: it's 8 bits—or, alternatively, 1 byte.

we'll explain how later.

2.2. Decomposed Color

For ordinary non-image work you almost always want TrueColor so you can make any color you want on the graphics output; you don't want to be restricted to 256 colors. In TrueColor, you have 256 different possibilities for each of the three colors RGB, so you have 256^3 color possibilities. It's definitely *not* a 256-element colortable! In IDL, you can turn color decomposition on with (guess what!)

```
device, decomposed=1.
```

2.3. What Have I Done?

You can oscillate between decomposed and combined color at will, to your heart's content. In fact, you sometimes need to do this when making color images in PostScript files. To see what you've told IDL, enter

```
help, /device
```

If you are in TrueColor with decomposed color turned off (i.e., a 256-element colortable) it will say "Graphics pixels: Combined"; if it isn't turned off it says "Graphics pixels: Decomposed". It also shows the sizes of all graphics windows and tells you which visual class you're running (TrueColor or DirectColor). If you are in PostScript mode, in which it writes all output to a PostScript file instead of the screen, it will tell you so.

3. VISUAL CLASSES: STATIC (TrueColor) VERSUS DYNAMIC (DirectColor)

When working with images you often want to have real-time cursor control of the colortable so that you can tailor the image appearance to your needs by changing the contrast or the displayed intensity range. For this you need a *dynamic* color table. TrueColor, and colortables based on TrueColor, do *not* have this cursor-control capability because TrueColor is a *static* colortable. The dynamic color table is called DirectColor. It works exactly the same as TrueColor, except that the transfer function between the data numbers and displayed intensities can be changed in real-time with the cursor. These days, the dynamic capability is not available on any machine running Windows or on Macs; it's available *only* on PCs running Linux. If you have such machine, you can try out DirectColor using our example below.

If you want a dynamic colortable and you are lucky enough not to have a Mac or PC with Windows, then you have to tell IDL what visual class you want—i.e., TrueColor or DirectColor—

and you must choose the mode first thing. Once you set it, you can't change it without re-entering IDL. With our locally-preferred startup file you are queried for this choice. Normally, you choose TrueColor: the only exception is when you want to manipulate image contrast and brightness with the cursor.

4. LET'S TALK ABOUT COMBINED COLOR—256-ENTRY (8-BIT) COLOR TABLES

The remainder of this tutorial assumes you have turned color decomposition off, i.e. you are using combined color. In this PseudoColor mode, the maximum number of color/intensity combinations that can be displayed simultaneously is 256. Therefore, images are represented by numbers that range $0 \rightarrow 255$. For this reason, displayed images are *always* represented by byte arrays. You can display other array data types, but IDL will convert whatever you give it to a byte array before displaying it. Therefore, if you display, say, an integer array (integers are two bytes long and range from $-32768 \rightarrow 32767$), and if numerical values in this array exceed 255, then the resulting image display will look weird. That's because, in converting from integer to byte, numbers that exceed 255 will "wrap around". For example, integer 255 equals byte 255, but integer 256 equals byte 0, integer 257 equals byte 1, etc. Below, we'll deal with these conversions in more detail.

For now, let's restrict our attention to black/white images—otherwise known as "grayscale" images. Gray, or white, is composed of an equal mixture of red, green, and blue, and all we deal with is the intensity I. In a grayscale image, the intensity of each pixel is related to the data value d in that pixel. Let's think of large intensity being white and small intensity being black; there are 256 different possible intensities, so I can range from $0 \rightarrow 255$. Similarly, the data values d can range from $0 \rightarrow 255$.

An important concept is the relationship between I and d. This is known as the *color table.* It specifies the mapping between data value and color/intensity—or, for a grayscale image, the mapping between data value and intensity.

4.1. Linear Mapping, both Direct and Reversed

The simplest mapping between data value d and intensity I is a linear one with

$$I = d \tag{1}$$

In this case, a data value $d = 255$ gives white and $d = 0$ gives black. This *direct mapping* is the default manner in which images are displayed on the computer screen: there is a black background on which the image is painted with increasing data values being increasingly white. However, on a piece of paper the relationship is usually reversed, because paper is white and provides a naturally white background. Thus, in this *reversed mapping*, we want to paint the image with increasing data values being increasingly black. This is also a linear mapping, but reversed:

$$I = 255 - d \tag{2}$$

NOTE: Printed images usually look *much* better with the reversed mapping, because printers have a hard time giving a uniformly black area with no streaks. This is the *first* reason why printed images should be made with a reversed mapping. The *second* reason is that in scientific journals, images with the reversed mapping are reproduced much better. The *third* reason is that making the paper black uses lots of printer toner, which is expensive. To reverse the color table, you can use equation 2. Alternatively, for a byte array called `img`, you can type `tv, not(img)` instead of `tv, img`.

4.2. Nonlinear Mapping

The linear mapping is often not very useful because you usually want to highlight weak features or bright features; we'll see an example below. The most commonly used nonlinear mapping uses a power law (this is the photographer's "characteristic curve") together with a "stretch", which cuts off the image at dim and bright intensity levels:

$$I = 255 \left(\frac{d - d_{bot}}{d_{top} - d_{bot}} \right)^{\gamma}, \ d = d_{bot} \rightarrow d_{top} \tag{3a}$$

$$I = 0, \ d \leq d_{bot} \tag{3b}$$

$$I = 255, \ d \geq d_{top} \tag{3c}$$

In a reversed mapping, you'd substitute $(255 - d)$ for d in the above equations.

There is one other commonly used nonlinear mapping, the so-called "histogram equalization" technique. In this technique, the mapping is modified on an image-by-image basis so that, all of the 255 colors are used in an equal number of pixels. Read about it in IDL's documentation on `hist_equal`.

4.3. Why We Need Nonlinear Mapping

Never forget that the idea is to turn the data array into an image that conveys information to the brain. The idea is *not* to be so strictly quantitative that details of interest are obscured.

You want to bring out details of interest. For example, for many images of the interstellar gas you want to emphasize weak structures at the expense of the fidelity gained by a strict proportionality between image brightness and data value. To this end, choose a color table and experiment with the image transfer function. At minimum, this involves changing the span of the data values represented in the image and raising the data values within that span to a power: a power less than unity to emphasize weak features, larger than unity for strong ones.

5. DISPLAYING AN IMAGE IN IDL

We are still assuming that you have turned color decomposition off, i.e. you are using a 256-entry color table. To accomplish this feat, see §2.

Below, we describe two methods to put the image on your computer screen (in computerese: "write the image onto your X window"). One (§5.2) is simpler and uses Tim Robishaw's (TR's) `display` procedure. The other uses IDL's native procedures; it's a bit more cumbersome but is more useful when you want absolute knowledge of which pixel is which.

5.1. An Example: Read the Image from a FITS File

First, generate an image. For this example there's a nice image of the X-ray sky, obtained by the ROSAT satellite on the web at:
`http://astro.berkeley.edu/~heiles/handouts/rass_c.fits`.
Copy this file to the disk area where you are running IDL. To accomplish this, go to
`http://astro.berkeley.edu/~heiles/handouts/` and hold Shift down while you left click on the file name; or right click on the file name. Easier for the Linux user is `wget`: type `wget http://astro.berkeley.edu/~heiles/handouts/rass_c.fits`

This file is in a format called "fits" format, which is the same format of many astronomical images. To read the data file into an array called `image`, it is easiest to use the IDL procedure called "readfits", which resides in the Goddard IDL library which, in turn, is already in your IDL path. All you have to do is type

```
image = readfits('rass_c.fits', headerinfo)
```

This returns two arrays: the image array (`image`) and information about the image (`headerinfo`); type `print, headerinfo` to see the header information. Now type `help, image` and IDL will tell you that it is a 480 × 240 FLOAT array.

Now we're ready to write the image to the X window.

5.2. Displaying and Examiming the Image with Tim Robishaw's (TR's) `display`

This is really easy. Here, you just type

```
display, image, out=out
```

and it displays the image in the current graphics window, automatically doing the size scaling and, also, the byte scaling of the `bytscl` command below. If you want a bigger image on your window, make a bigger window and invoke `display` again. The variable `out` is output from the procedure and is a structure which contains useful information for later processing.

`display` has other features/options and can produce labelled axes, enhanced contrast, etc. See §2.1 of TRDIDL. In addition, you can interactively examine data points with the cursor using `trc`, and you can plot 1d cross sections versus x or y using `trp`. See §2.3 of TRDIDL.

5.3. Displaying and Examining the Image with IDL's Native Procedures

This method is more cumbersome, but you can retain the original pixels, so you always know how your actions relate to the original pixels.

5.3.1. Image Size and Window Size

This image is 480×240. Each element will occupy a single pixel on the screen, so we need a window of at least that size to display the whole image. We can use a bigger window, in which case the image won't fill the window area. If we use a smaller window, only part of the image will be displayed. We can create a window of the appropriate size, that is with numbers of pixels equal to the same dimensions of the data array, by typing

```
window, xsize=480, ysize=240
```

and then displaying as directed in §5.3.2.

Maybe you'd like a bigger image in a bigger window so you can see things more clearly. Or maybe the image is too large for your screen and you need to make it smaller so it fits. In either case, you need to change the size of the image as measured in pixels. IDL does this easily. Suppose you want to increase the size by a factor of 2 in the horizontal and 3 in the vertical direction[2], i.e. to make an array of size 960×720. Do this by

```
bigimage = rebin(image, 960, 720)
```

[2]Using different factors for horizontal and vertical changes the aspect ratio, which is usually a bad idea; we do it here simply for illustration.

Then create an appropriately-sized window (e.g. with `window, 5, xsize=960, ysize=720`); this creates a new window, numbered 5, and leaves the old ones in place.

You can also resize the image using `congrid`, which works for non-integral factors. You might say, "Well, I'll always use `congrid`—it's more flexible". *But be careful!* `congrid` and `rebin` handle enlargement and ensmallment differently, and treat the edges differently. With `congrid`, you *almost certainly* don't want to use the default options; look carefully at the keywords and try them out on a short 1-d array to see their effects. Usually, `rebin` is better; don't use `congrid` unless you know what you're doing.

5.3.2. Displaying the Image with `tv`

Display the image by typing

```
tv, image
```

and you see a gray mishmash oval. The oval is the Aitoff projection of the entire sky in soft X-rays. The mishmash occurs because the data values in `image` exceed the allowable $0 \rightarrow 255$ range of a byte array, so there's lots of wrapping. You can use the `max` and `min` functions (or, nicer, Goddard's `minmax` function) to determine that the data values range from about $-174 \rightarrow 45337$, thus far exceeding the valid range for a byte array.

You can scale the data so that they all fit in the allowable byte range $0 \rightarrow 255$. We'll first produce a byte array, which we'll call `byteimage`, from `image`...

```
byteimage = bytscl(image)
```

This linearly scales `image`, which ranges $-174 \rightarrow 45337$, into `byteimage`, ranging from $0 \rightarrow 255$. To display this image...

```
tv, byteimage
```

At this point, you've got the same image you'd have at the end of §5.2 by using `display`.

5.4. Using `rdpix` and `tr_profiles` (or `profiles`) After You Used `tv`

If you used `tv` and not `display`, then you can use `rdpix`, which prints the pixel values of the image you move the cursor on the image. What we really want is the values of the *original* data array, not its byte counterpart, so we specify that by typing...

```
rdpix, image
```

and then the printed numbers will be of the *original* data array `image` (If you want the byte image, use `rdpix.), byteimage`. After some inspection we see that limiting the data range to $0 \rightarrow 2000$

would indeed be a good start.

Nicer than `rdpix` is the native IDL procedure `profiles`. Nicer still is TR's enhanced version, `tr_profiles`. If you used `tv`, type

```
tr_profiles, image
```

and follow the printed instructions. You get a plot of the original data array in its original units, either in the horizontal or vertical direction, along a line you choose with the cursor. In our example, this plot is so compressed that it is virtually worthless, because the plot automatically scales to the minimum and maximum values of the array; you can get around this easily by using the $<$ and $>$ operators; for example,

```
tr_profiles, (0 > (image < 5000))
```

Here, the (`image` $<$ `5000`) means "take whichever number is smaller, either the data value in `image` or the number 2000"; and the `0` $>$ `X` means "take whichever number is larger, either the data value in `X` or the number 0". So the plot yaxis range is limited to the range 0 to 5000.

6. EXPLORE AND HAVE SOME FUN WITH THIS IMAGE!

What Do You See in this Image? All you see is two white dots! These two dots are the strongest X-ray sources in the sky—the one on the left is a point source called "Cygnus XR-1", and the one on the right is the Vela supernova remnant, home of the famous "Vela pulsar". In supernova remnants, the X-ray emission is produced by hot, $\sim 10^6$ K gas heated by the expanding shock of the supernova remnant.

These images contain much more! To see more, we need to change the contrast by invoking a nonlinear mapping of data d to screen intensity I. You can do this dynamically (in real time with the cursor; see §6.1) if you have DirectColor, and you can do it statically (see §6.2).

There's lots more in image processing and display. Of course, you can manipulate images mathematically, just as you can any other IDL variable or array—but remember to manipulate the *original* array instead of its *byte* counterpart. As we did above, you can play with color tables with `xloadct` and `ct_fiddle`. You can make a histogram [e.g. `histo = histogram(image)`] of the original image; this tells you where most of the brightness data are concentrated. And if you're interested in only a *portion* of the image, you can select this portion with the cursor using `defroi`. You can do "histogram equalization" with `hist_equal`. You can rotate with `rotate` or `rot`, `transpose`, `zoom`, draw `contours`, label your images and make coordinates using `plot` (with the `/noerase` keyword) and `xyouts`, etc., etc., etc.

To learn the range of native IDL capabilities, enter IDL and type "?", which brings up the online help window. Click on "List of IDL Routines by Functional Area" and take a look at the sections entitled "Array Manipulation", "Color Table Manipulation", "Direct Graphics", "Image

Processing", "Plotting", "Signal Processing".

6.1. If You Have a Dynamic Colortable, Try the Cursor...

If you're in DirectColor[3], try changing the contrast with the cursor. The particular nonlinear mapping of equation 3 can be invoked easily and automatically in IDL by typing `xloadct` or—better for interactive work—Tim Robishaw's `ct_fiddle`[4]. These programs allow you to change the values of $(\gamma, d_{bot}, d_{top})$ with the mouse and watch the contrast of your image change.[5] You can also change from grayscale to PseudoColor by selecting one of IDL's predefined color tables, using either `xloadct` with the mouse or manually using `loadct` (e.g., `loadct, 5` loads IDL's color table number 5. You can, of course, define your own colortable.).

Using either `xloadct` or `ct_fiddle`, play around with contrast. You can see that the sky contains a weak, diffuse glow in X-rays. Trouble is, though, that this glow is so weak that the intensity (I) values of this glow all lie in the range $d \sim 0 \rightarrow 4$. This provides very little dynamic range for this glow, so we need to expand this range by changing the colortable so that we can see its structure more clearly.

6.2. Manipulating the Image by Manually Changing the Colortable

To see anything other than the two bright sources, we need to change one or more of $(d_{bot}, d_{top}, \gamma)$. How much should we expand the dynamic range? We might make a guess and try $d_{bot} = -174$ and $d_{top} = 2000$. If that didn't give a nice result, we could try some other values. But we don't have to guess! IDL provides several nice ways to interactively print the values of the image.

Firstly, you can get a quick feel for the interesting data range by just doing `plot, image` and visually estimating the range of interest. However, you learn more by sampling the image itself using `trc` and `trp` (if you used `display`, or `rdpix` and `trprofiles` (if you used `tv`).

---

[3]With a dynamic colortable, sometimes the whole terminal screen changes colors when you move the cursor onto an IDL window. This is commonly known as *flashing*. This is one reason why we recommend using TrueColor for all IDL sessions in which you will not be doing image processing with dynamic color tables. There can be no flashing with TrueColor because there are no dynamic color tables.

[4]If you are using a dynamic color table, `ct_fiddle` will change the image display in real time for any specified combination of R, G, and B. However, with `xloadct` the image doesn't track the sliders; after you move a slider, you need to put the cursor on either the colorbar in the `xloadct` GUI or on the image itself.

[5]`xloadct` also allows you to select a reversed mapping and to select a multitude of predefined color tables, not only grayscale but many others. `xloadct` also allows you to generate any completely arbitrary nonlinear mapping; click on "Function" and experiment.

7. REDISPLAY USING THE MODIFIED COLORTABLE

Now, having determined suitable values for d_{bot} and d_{top}, we want to display the appropriately-scaled image.

7.1. Redisplaying the Image with `display`

If you're using TR's `display`, it's just

```
display, image, min=0, max=2000
```

7.2. Redisplaying the Image with `tv`

To display the data range $0 \rightarrow 2000$, we again use the `bytscl` command as above but limit the data range by typing...

```
byteimage = bytscl( image, min=0, max=2000) .
```

This performs a modified scaling, mapping the original data range $0 \rightarrow 2000$ into the byte value range $0 \rightarrow 255$. It also sets any original data numbers that exceed 2000 equal to 255, and any that are smaller than zero equal to zero—so it obliterates information on the strongest features.

Now display this with

```
tv, byteimage
```

8. AND NOW THE ASTRONOMY...

If you have DirectColor, Use `xloadct` or `ct_fiddle` to play around with contrast; if not, experiment with `gamma`. When you increase the sensitivity to small numbers by using $\gamma < 1$, you see the diffuse background. See that huge circular structure in the middle? That's the "North Polar Spur". It occupies an angle of about 120°. It's close—almost touching our noses! It's caused by several dozen supernovae that have exploded, producing a "superbubble". These supernovae were located in the large cluster of young stars in the Scorpio constellation—some of the stars you see there on a dark night will explode as supernovae some day, adding to the energy stored in the hot gas and brightening the X-ray emission. You also can see a bunch of fairly weak point sources and other diffuse structures.

9. ADD A COLORBAR! USING TIM ROBISHAW'S `colorbar`

Displaying an image without a colorbar is like displaying a graph with axes that have no numbers or labels. It's easy to add a colorbar with Tim's `colorbar`. This procedure does either 1d or 2d colorbars; see the images in the handout "1d2d3d: One, Two, and Three Dimensional Color Images". Here's how:

1. Decide where to put the colorbar. Decide if you want a horizontal colorbar (usually on the top) or a vertical one (usually on the right hand side). There's a keyword `vertical`; set it appropriately.

2. Define `position`, a 4-element vector that specifies the normalized coordinates[6] of the corners of the colorbar. The 4 elements are in order (lowleftx, lowlefty, uprgtx, uprgty). For example, for a horizontal colorbar on the top you might choose something like `position = [ 0.1, 0.8, 0.9, 0.9]` . If you have a horizontal colorbar and you want it to extend the exact length of the image, you can get the normalized position of the left and right edges of the image from the system variable !x.window; similarly, the top and bottom are stored in !y.window. So if you wanted a colorbar that matched the width of the image and you wanted it to have a height of 10% of the height of the window, then you could make the colorbar float 5% above the top of the image by providing a position vector of:

   ```
   position = [!x.window[0],!y.window[1]+0.05,!x.window[1],!y.window[1]+0.05+0.10]
   ```

   You need enough space for the colorbar. If any element of the `position` vector lies outside the range $0 \rightarrow 1$, then when you run `colorbar` it will spill over the edge of the window, in which case you might get an error like this:

   ```
   % DISPLAY: Normalized POSITION[2:3] must be less than 1.
   % Execution halted at: DISPLAY          731
   ```

9.1. A 1d Colorbar

For an example, see the appendices in our tutorial "1d2d3d: One, Two, and Three Dimensional Color Images".

1. Define the data values that correspond to the min and max of the image brightness; for the description of `display` above, these would be `min` and `max`. Set `crange=[min, max]`.

2. If you raised the data to a power (`gamma`) to change the contrast, then you need to tell `colorbar` by setting `cgamma = gamma`. If you didn't use `gamma` on the data, then set `cgamma=1` or leave it undefined.

[6]Normalized coordinates range from 0 to 1 and are the fraction of the size of the window.

3. Define the name of the displayed quantity, e.g. `xtitle='Xray intensity'`.

4. Then make the colorbar:

```
colorbar, POSITION=position, CRANGE=crange, CGAMMA=cgamma, $
        NEGATIVE=negative, VERTICAL=vertical, XTITLE=xtitle
```

   For more info on `colorbar` parameters, see its documentation.

9.2. A 2d Colorbar

The easiest way to make a 2d image with its colorbar is with our `display_2d.pro`; this does everything with one procedure call, but not much flexiblity. See §10 and, for an example, see the appendices in our tutorial "1d2d3d: One, Two, and Three Dimensional Color Images".

If you want to do it all yourself:

1. For a 2d image you need to specify the min and max for both the color image (`cmin, cmax`) and the intensity image (`imin, imax`). These min/max values are specified as 2-element vectors. So for the color image, it's `crange= [cmin, cmax]` and for the intensity image it's `irange= [imin, imax]`. Similarly, each image has its own gamma (the power to which the data numbers are raised): `cgamma` and `igamma`. It should be obvious that these gamma values should correspond to what you used when you displayed the image. The titles on the colorbar are `xtitle` (the name of the variable that represents color, e.g. velocity) and `ytitle`.

2. For a 2d color image and colorbar you are using a 256-entry colortable to define the colors. The 256 (r,g,b) intensities are represented with a 256×3 array called `colr`. If you use the physiologically-correct colortable discussed in the handout "1d2d3d: One, Two, and Three Dimensional Color Images", then you get this array from the procedure `pseudo_ch, colr`. We assume this in our example just below; if you get `colr` in another way, you still need to `loadct,0` before calling `colorbar`.

3. Finally, invoke the colorbar procedure:

```
pseudo_ch, colr
loadct,0
colorbar, POSITION=position, RGB=colr, $
        CRANGE=crange, IRANGE=irange, CGAMMA=cgamma, IGAMMA=igamma, $
        NEGATIVE=negative, VERTICAL=vertical, $
        XTITLE=xtitle, YTITLE=ytitle
```

N.B. Any of the keywords that you can send to `plot` are also accepted by `display` and `colorbar`. As an example, sometimes it's easier to see tickmarks on an image and on a colorbar if they are pointing outwards; this could be accomplished by passing the keyword `ticklen=-0.02` to both `display` and `colorbar`.

10. USING `display_2d` FOR 2D IMAGES

For an example, see the appendices in our tutorial "1d2d3d: One, Two, and Three Dimensional Color Images".

This is a quick and easy way to make a 2d image with a colorbar. It is well-enough documented. A restriction is that the colorbar is always on top. The only tricky part of using this is making the image and colorbar fit into the window in a nice way. This fit is controlled mainly by the window size, and also by `cbar_posn`, which is the four-element vector that specifies the colorbar placement in normalized coordinates. Usually you can just adjust that window size and use the default for `cbar_posn`, which is automatically set if you don't specify it.

Controlling the window size on the X window is a matter of specifying its size in pixels. For example, when you open a window, say number 5, you type `window, 5, xsize=800, ysize=600` (for an 800×600 window). For a PS window, you control the size with the `xsize` and `ysize` parameters in the `psopen` command.

11. USING `rgbimg` FOR 3D COLOR IMAGES

We have a clumsily-written but reliable procedure to produce a 3d image with a fancy colorbar like those of Figure 6 in the handout "1d2d3d: One, Two, and Three Dimensional Color Images". A severe restriction is that the image must have dimensions 541×541; you can use `congrid` to convert your image to this size (but see the cautions about `congrid` in in the handout "1d2d3d: One, Two, and Three Dimensional Color Images"). The procedure is called `rgbimg.pro` and it is reasonably well-documented.

It is a pleasure to thank Tim Robishaw for teaching me a lot, providing software, and constructive comments.