COLOR IMAGES TO REPRESENT ONE, TWO, AND THREE DIMENSIONAL DATA
May 7, 2002

Carl Heiles

Before reading this, look at *BIDIDL*. This discusses the basics of image display and also tells how to make poscript files of images using our home grown routines **hardimage** and **openimageps**. You can also make tiff files, gif files, etc; see IDL's help facility under *scientific data formats*.

This handouts discusses the use of color in representing arrays of data. Normally we think of image display as just that—a way to display an image. However, you can combine more information into a single image by the combined use of color and intensity. Using color for these purposes is great, but using it intelligently is not trivial. There are three basic uses for color, and below we discuss each in the correspondingly-numbered section:

(1) Using a color table to represent a single quantity. Consider this as a *one-dimensional* mapping of the quantity into color. The most common example is the ordinary greyscale image, in which the degree of whiteness or blackness represents the value of the quantity represented in the image (such as light intensity). Many astronomers like to replace the simple greyscale with a colorscale because it's easier to discern changes of color than small changes of brightness. However, this has its problems: firstly, 20% of the human population is at least partly colorblind; secondly, color reproductions in printed journals are very expensive.

(2) Using color to represent the value of a single quantity, such as the mean velocity of a spectral line, and using the lightness (brightness of the color: black is zero, full brightness is maximum lightness) to represent a different quantity, such as intensity of the spectral line. This is a *two-dimensional* mapping onto the image plane. Here, one uses a color table to represent one variable, as one sometimes does in one-dimensional mapping, plus the addition of the change in lightness to represent the second variable.

(3) Using color to represent the values of three independent quantities, with the lightness of each color proportional to its quantity. For example, using red for IR, green for optical, blue for X-ray. This is a *three-dimensional* mapping onto the image plane. You can't represent more than three dimensions in a color image because the eye is responsive to only three independent colors.

Finally, there are some related matters such as overplotting onto images and calibration; we discuss these in section **(4)**.

1. ONE-DIMENSIONAL COLOR: THE 8-BIT COLOR TABLE

1.1. Pseudocolor

Pseudocolor mode is the ideal one for the one-d color case because the color table allows you to specify 256 different color/lightness combinations, one for each image intensity. An important consideration is the *number of colors*. Of course, you normally invoke this *8-bit pseudocolor mode* by typing

device, pseudo=8

as the first thing you do in IDL. After doing this, you might think you have 256 different colors available on your IDL display. But you're wrong!

As discussed in *BIDIDL*, for the X window output device (which are the windows in which you make plots and images) IDL doesn't have all 256 combinations available to it because the operating system takes up some colors for itself. However, for the *postscript* output device IDL *always* has 256 combinations available to it. Thus the screen has fewer colors available to IDL than the postscript device does. Thus, the conversion of images to PS needs to interpolate the screen's colors to fill the 256 colors of postscript. This can lead to distortions in the color appearance; usually these are small, but not always.

To avoid these problems, it's most reliable to *force* IDL to use all 256 colors for the X window displays; that way, once you generate an image on the screen, you know exactly how it will come out in the postscript file. If you do this, then the screen colors will "flash" depending on whether your cursor is on an IDL window or not (see *BIDIDL*).

If you're running IDL with 256 colors, you'll find that the contrast sliders in **xloadct** don't work as you expect. You can duplicate the operation of these sliders using our home-grown routine **diddle**: the left-hand mouse button is like **xloadct**'s *Stretch Bottom*, the right-hand button like *Stretch top*, and the middle button like *Gamma Correction*; moving the cursor horizontally within the small window changes the quantities, and you get out of **diddle** by hitting a key on the keyboard.

Once you find the combination of stretch and gamma that makes the image look good, we advise that you apply these to the *data* and produce an image using stretch $0 \rightarrow 255$ and $gamma = 1$. That way you will know for sure that your PS file will look very much like what you have on the screen.

2. TWO-DIMENSIONAL COLOR: AN 8-BIT COLOR TABLE WITH 24-BIT COLOR

Examples: */dzd2/heiles/courses/handouts/images/xmxcstd1.ps.*

2.1. Truecolor and Directcolor

For two- and three-dimensional color you need the 24-bit color display, which allows you to generate any color/lightness combination. Your machine might display 24-bit color. However, there are two flavors of 24-bit color: *Truecolor* and *Directcolor*. When you enter IDL, it will choose the most advanced flavor; if you are lucky enough to have Directcolor, that's what you'll get. We discuss these color modes more extensively in *BIDIDL*.

In Truecolor, there are no color tables. This means that it is impossible to interactively change the contrast of an image using **diddle**. If you run IDL under (God Forbid!) Windows 98, you'll find that Truecolor is the only mode you've got. This is fine for two-dimensional color applications, and this is what you will probably want to choose on your UNIX machine. To do this, the first thing you type in IDL should be

device, true_color=24

If you now type **help, /device**, the display will tell you that the Translation table is bypassed. You have a full 256 levels of intensity for each of the three colors. This absence of an IDL colortable means no cursor control of image contrast—and no flashing.

2.2. Choosing a color table for the first dimension

Suppose you are representing the mean velocity of a spectral line by color. To do this, you should generate a 256 entry color table in which each color represents velocity. You might think that a natural choice would be the spectrum, from red to blue, representing positive to negative velocities.

However, this is an exceedingly poor choice because you wish to have a second dimension, namely to represent the line intensity by the color's *lightness*, which is apparent brightness of the color. And the human visual system has a very low sensitivity to blue light. Therefore, the brain will perceive all negative velocities, represented by blue, as less intense than the corresponding positive velocities, even though the line intensities are the same or larger.

In other words, you need to account for human physiology in the choice of the color table. There has been lots of research on this topic and IDL provides three procedures that are tailored to meet these needs; see IDL's **?pseudo** help. I've found that a good colortable is given by

pseudo, 100, 100, 100, 100, 22.5, .7, colr

The **pseudo** procedure creates a color table based on *lightness* (apparent brightness), *saturation* (degree of color purity: more white mixed in means a less pure color, one that looks more nearly pastel), and hue (color). **pseudo** loads the color table, firstly into the 2-d array *colr[256,3]*; and secondly into the color table of IDL's *colors* common block (not that this matters for Truecolor

mode!).

This color table runs from magenta to pale blue to green to yellow to red. The array *colr* has 256 elements in its first dimension, corresponding to 256 velocities. The second dimension contains three elements, corresponding to the three colors red, green, blue. Suppose, for example, that in a particular pixel the velocity is such that this first index is 86. Then *colr[86,0]* is the intensity of red, *colr[86,1]* that of green, and *color[86,2]* that of blue for that particular velocity. The velocity 86, being about $\frac{1}{3}$ the way from magenta to red, corresponds to roughly pale blue. These colors all appear to be roughly the same intensity to the average human.

2.3. Changing the lightness for the second dimension

In the aforementioned pixel, the color is given by the mixture of red, green, and blue specified by *colr[86,*]*. Displaying the colors at full intensity gives maximum lightness. You reduce the lightness by scaling all colors simultaneously. For example, suppose the maximum line intensity is **lmax** and the minimum is zero. Let **lpix** be the line intensity in the 2-d image array and *redimg, grnimg, bluimg* be the three 2-d image arrays that you will generate and pass to the **tv** routine. Then, for the whole image, the IDL statements

redimg = byte(colr[*,0] * (lpix/lmax) > 0 < 255)

grnimg = byte(colr[*,1] * (lpix/lmax) > 0 < 255)

bluimg = byte(colr[*,2] * (lpix/lmax) > 0 < 255)

generate the arrays. We convert to byte, although that's not strictly necessary because IDL will do it automatically. What *is* strictly necessary is to keep the numbers ≥ 0 and ≤ 255, which is what the $(> 0 < 255)$ ensures.

2.4. Writing the image to the display

You write the image to the display using **tv** (*not* **tvscl**: you want to preserve brightness ratios!). You have to write three images, one for each color.

For the screen there are two ways to do this. You can use the convenient form, which works *only for X*

tv, redimg, channel=1

tv, grnimg, channel=2

tv, bluimg, channel=3

or you can use the same format that postscript requires.

The other way fossssssr X, and the way required for postscript, is

tv, [[[redimg]], [[grnimg]], [[bluimg]]], /normal, true=3, $
ybotm, xbarleft, ysize=xbarsize, xsize=yplotsize

where on the second line we've included some extra keywords to position and size the image; you should look at the documentation for **tv**. The **normal** means that positions are specified in normalized coordinates, which is always a good idea when you will end up making a postscript file; see *BIDIDL*.

2.5. Adjusting the image parameters

The colors span a range of line velocity, and you need to choose this range. Similarly, the lightnesses span a range of line intensity, and you need to choose this properly and, in addition, possibly need to change the contrast (the "gamma factor"). To do this in this two-dimensional color mode, you need to change the parameters and rewrite the image to the screen; **diddle** won't give good results because it operates on all three colors identically, which isn't what you want (which is why you might as well use Truecolor for two-dimensional color). You will probably want to write some software that allows you to use the cursor to interactively change the ranges while you look at the image so that you can judge what ranges are best. I have some poorly-documented software that I might be willing to share with you if you bribe me with a nice enough treat.

3. THREE-DIMENSIONAL COLOR: WITH 24-BIT COLOR

Examples: */dzd2/heiles/courses/handouts/images/jerry_rgb1.ps, itp1.ps, itp2.ps.*

Here you want to represent three independent quantities by three independent, orthogonal-to-the-eye colors. You can use two color schemes. One is the obvious red, green, blue; these correspond directly to the three cones of sensitivity in the eye. Alternatively, you can use their complements: cyan, magenta, and yellow. Recall that $white = (red + blue + green)$, so you can generate a second set of eye-orthogonal colors as $cyan = (green + blue)$, $magenta = (red + blue)$, $yellow = (red + green)$. The former scheme is susceptible to the poor blue-sensitivity of the human visual system, but sometimes this is an advantage when one of the images is less important. Experiment for each individual case!

3.1. Directcolor and diddle: adjusting the image parameters

This is the ideal application for *Directcolor*. In Directolor, each color has its own colortable and you can manipulate them independently using our home-grown procedure **diddle**. Therefore,

you can independently change one image relative to the others and achieve the best balance. If your machine doesn't have Directcolor and you are doing three-dimensional color, you'll save lots of time by going to a different machine that has Directcolor.

In Directcolor, IDL takes control over all of the colortables. In other words, IDL forces a private colortable. This means you'll get the "flashing" when the cursor moves on and off of IDL's display windows; see our above discussion in section 1.

diddle can work on any combination of colors. For example, to change the red image alone, type **diddle, 'r'**; to change the cyan image, type **diddle, 'gb'**; to change all three, type either **diddle, 'rgb'** or just **diddle** (because all three colors is the default).

4. MORE

4.1. Overplotting a coordinate grid

Often you wish to overplot an image, for example to make a coordinate grid or place a point to label an object. To make an overplot, use **plot** with the appropriate keywords **position** (which places the plot borders where you specify), **normal** (best to use normalized coordinates), and **noerase** (which doesn't erase before overplotting); see the details in *BIDIDL*. If the labels of the plot extend outside the image, then you have to be careful with the labels' colors: on the screen they are written on the *black background*, but on the ps device they are written on a *white background*.

Let's reiterate that point. The default background for X is black, while that for PS is white. When you are making a pretty picture to publish, you might find it useful to give the X window a white background. You can do this by **tv**'ing a white image onto the whole window immediately after you create it.

After you make your image overplotted with a plotting grid, you can use **plots** to plot a single symbol somewhere or **xyouts** to write a label. However, when you use these you should use either *data* or *normalized* coordinates, not *device* coordinates, because if you use the same code to generate a postscript file the device coordinates are meaningless.

For more details see *BIDIDL*.

4.2. Annotating with color in truecolor and directcolor

When you annotate, for example with the **plot** or **xyouts** commands, you can make colored characters and plot colored lines using the **color** keyword. **Unfortunately, the use of this keyword differs greatly between X and PS**.

4.2.1. Annotating with color in X

In 24-bit color X, the color is specified by a 24-bit (3-byte) word. This is true both for the image and for characters written by, say, **xyouts**. The least significant byte is red, the next one green, and the most significant blue. Thus, pure red is 255; pure green is 255*256; pure blue is 255*256*256. White is $red + green + blue$, magenta is $red + blue$, etc. So if you want to annotate with green letters centered at data coordinates (5,6) you type **xyouts, 5, 6, 'This is written as green', color=green, /data**, where **green** $= 255 * 256$. (Specifying **data** is usually not necessary).

4.2.2. Annotating with color in PS

Unfortunately, PS uses a different scheme. In PS you *must* specify the color of *text* with a *colortable*, whose length is limited to 256 elements. This is, of course, completely different from the way colored *images* are written. Here's a simple example of such a colortable that consists of just five colors: black, red, green, blue, and white. This is a 5-element colortable; you can have as many as 256.

First, generate the red, green, and blue intensities of the 5 colors:

```
red = 255 * [0, 1, 0, 0, 1]
grn = 255 * [0, 0, 1, 0, 1]
blu = 255 * [0, 0, 0, 1, 1]

which correspond to colors

color = [ black, red, green, blue, white]
```

and then load them into IDL's internal color tables

```
tvlct, red, grn, blu
```

Now you're ready to roll! You'd annotate with green characters using **xyouts, 5, 6, 'This is written as green!' color=2, /data**.

This necessity to use different color schemes in X and PS is a royal pain.

4.3. Colorbars

The first thing you learned in kindergarten should have been: "When you make a plot, label and quantify the axes". What are the scientific images we've been discussing except plots of one,

two, or three variables—except that the plots are in 2-d space. And the first thing you learned in kindergarten *still applies*. This means that you need to quantify the meaning of the intensity, or colors. You do this with a *colorbar*.

There is nothing more infuriating than an image in a so-called scientific journal whose intensities or colors are not quantified.

You do this calibration with a colorbar, which is a bar along which the color, or intensity, changes from one extreme to the other; labels quantify the colors. You can make your own colorbar using **tv** with the **X, Y** arguments to position the subwindow and the **xsize, ysize** keywords to set its size (except that, on the X window, the size is set by defining the number of pixels in the array displayed by **tv**). See, for example, Fanning page 234, or the *colorbar* procedure in his website *http://www.dfanning.com/*.

The above colorbars are fine for one-dimensional color. For two-dimensional color, you need to also include a change in intensity in the orthogonal direction. For this, you need to make your own. Or, again, I might be persuaded to give you my undocumented software that makes colorbars like that in *c2hihalfa.ps*.

Three-dimensional color needs more than a single colorbar, which can represent only two dimensions. I have attacked this with the scheme in */dzd2/heiles/courses/handouts/images/jerry_rgb1.ps*. Here, a bar across the top of the image is divided into eight squares. The three leftmost squares give the characteristic curves for red, green, and blue, and are labeled with the quantity depicted. The five rightmost squares exhibit how the colors mix when combined. In each square, blue has a different intensity, constant within the whole square, running from 0, 0.25 ... 1.0 of full intensity. Within each square, red is zero at the left and maximum at the right; green is zero at the bottom and maximum at the top. So the top right corner of each of these five squares is full red and green: the first square, with no blue, shows yellow and the last, with full blue, white.

4.4. Making postscript files

See *BIDIDL*. You can use **hardimage**, which reads pixels from the screen and writes them directly into a postscript file. Also, you can use **openimageps**; write your plot and character annotation; and then **closeps**. These routines work in all color modes.

For pseudocolor and directcolor follow this, the voice of experience. In principle, you can apply a colortable to the three-dimensional color images using **diddle** and then this colortable could apply to the ps image as well. But I don't recommend this technique. Rather, write a *new byte array* that is based on the parameters you obtain from **diddle** and display it with a linear color table—equivalent to using Truecolor. Then there is absolutely no possibility for screwup when transferring the data to postscript. X-windows and postscript seem to work a bit differently,

in my experience, and I've never understood why.

4.5. Displaying 24-bit images with utility routines such as xv

On Sun machines, XV doesn't work very well on 24-bit color PS images (those that are generated with truecolor and directcolor). I don't know why. Use the UNIX command **display** instead. On my linux laptop, both work fine.