ONE, TWO, AND THREE DIMENSIONAL COLOR IMAGES
January 18, 2007

Carl Heiles

Contents

This is the fourth in our series of four tutorials on images. Before starting this one you should read our introductory overview on these four tutorials, which is entitled **FOUR TUTORIALS ON IMAGES: NOVICE → EXPERT**; and also the first three in the series.

This tutorial 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 256-entry 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 grayscale 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 grayscale with a colorscale because it's easier to discern changes of color than small changes of brightness. However, this has its problems: firstly, almost 20% of the human male population is at least partly colorblind; secondly, color reproductions in printed journals are very expensive.

(2) Using 256 *colors* to represent the value of a single quantity, such as the mean velocity of a spectral line, together with 256 values of *lightness* (brightness of the color: black is zero, full brightness is maximum lightness of a particular color) to represent a different quantity, such as intensity of the spectral line. You are using two visually orthogonal qualities—color and lightness—to represent two quantities, so 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 256 lightnesses 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 independently responsive to only three colors.

Finally, there are some related matters such as overplotting onto images and calibration; we discuss these in later sections.

1. A PRELIMINARY: DYNAMIC VERSUS STATIC COLOR TABLES

Modern computers work in 24-bit color mode, meaning that each of the three colors (R, G, B) can have 256 values and they are all independent. This means the total number of colors is 256^3 ("millions of colors"). With 24 bit colors there are two flavors, *TrueColor* and *DirectColor*. We discuss these color modes more extensively in **BIDIDL**.

TrueColor doesn't have dynamic color tables. This means that it is impossible to interactively change the contrast of an image using the cursor. Without this interactive behavior it is very difficult to optimally tune the image contrast. If you run IDL under (God Forbid!) Windows or Mac OSX, you'll find that TrueColor is the only mode you've got.

Linux and UNIX users can invoke DirectColor, which does have dynamic color tables—one for each color. In IDL, we Linux users invoke `ct_fiddle` to interactively tune the image contrast and color balance. If you are dealing with images and don't run Linux, you might want to persuade one of your Linux buddies to give up her machine for image processing work.

Our startup file forces you to choose among various color modes. If you don't use ours, you need to specify the mode before plotting anything. For example, if you want to invoke TrueColor, then 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; each color is represented by a 24-bit number. If instead you want a 256-color specification, called a color table, type

```
device, decomposed=0
```

You can do this at any time. You can convert back to the 256^3 method by typing `device, /decomposed`.

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

2.1. The Basic Idea

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, using the techniques discussed in **BIDIDL**. 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. Using `ct_fiddle` makes this easy on DirectColor displays, which have a dynamic color table; with the static color displays it's more cumbersome.

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. To generate PostScript files with nice annotations and fonts, use the procedures in the second and third memos in this series, entitled **MAKING ANNOTATED IMAGES (WITH COLORBARS TOO!)** and **MAKING, PRINTING, AND POWERPOINTING POSTSCRIPT FILES OF GRAPHS AND IMAGES**.

2.2. Considerations in Choosing a Color Table

For publication in journals, a grayscale color table is usually best because color reproductions are very expensive. However, for other purposes, such as lecture slides, a color table is often preferred because you can see more details.

I have, myself, not yet come upon the ideal color table. The ideal color table would:

1. Have a monotonically and smoothly changing color with intensity value.
2. Have no visual ambiguity: a non-colorblind person would never suffer any ambiguity about the relationship between color and intensity.
3. Have the most intense portions of the image be represented by the physiologically brightest color in the color table.
4. Have the least intense portions of the image *not* be represented by the physiologically dimmest color (which is blue), because it will not clearly show weak features against the black background of zero intensity.

Color Tables in common use do not satisfy all these conditions. For example, the very popular color table used by the VLA's AIPS uses red for the most intense parts of the image; the physiologically brighter white represents a lower intensity level. Despite this departure from a purist philosophy, the AIPS color table is quite good. Many other color tables that people use are simply awful when considered from a visual, not to mention logical, perspective.

In my opinion, IDL provides four color tables that approach the ideal. Their names and numbers are: Hue Sat Lightness 1 and 2 (numbers 19 and 20); Hue Sat Value 1 and 2 (numbers 21 and 22). (For both of these, number 2 needs to be reversed because white is at the low-intensity end of the range). It also provides another color table which doesn't satisfy these criteria very well but its colors are so striking that it is psychologically quite appealing (to me, at any rate); it's Standard Gamma II (number 5). You can load these color tables either from `xloadct` or (preferably) using the IDL command `loadct, nr` (where `nr` is the number of the color table given above). You can also make any color table you choose and load it with `tvlct`; see §3.1.

2.3. The Colorbar

There's nothing more annoying than seeing a nice one-dimensional image without an indication of how the grayscale or color relates to data value. You do this with a colorbar. Shame on you for even *thinking* of generating a 1-d image without a colorbar! See §5.

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

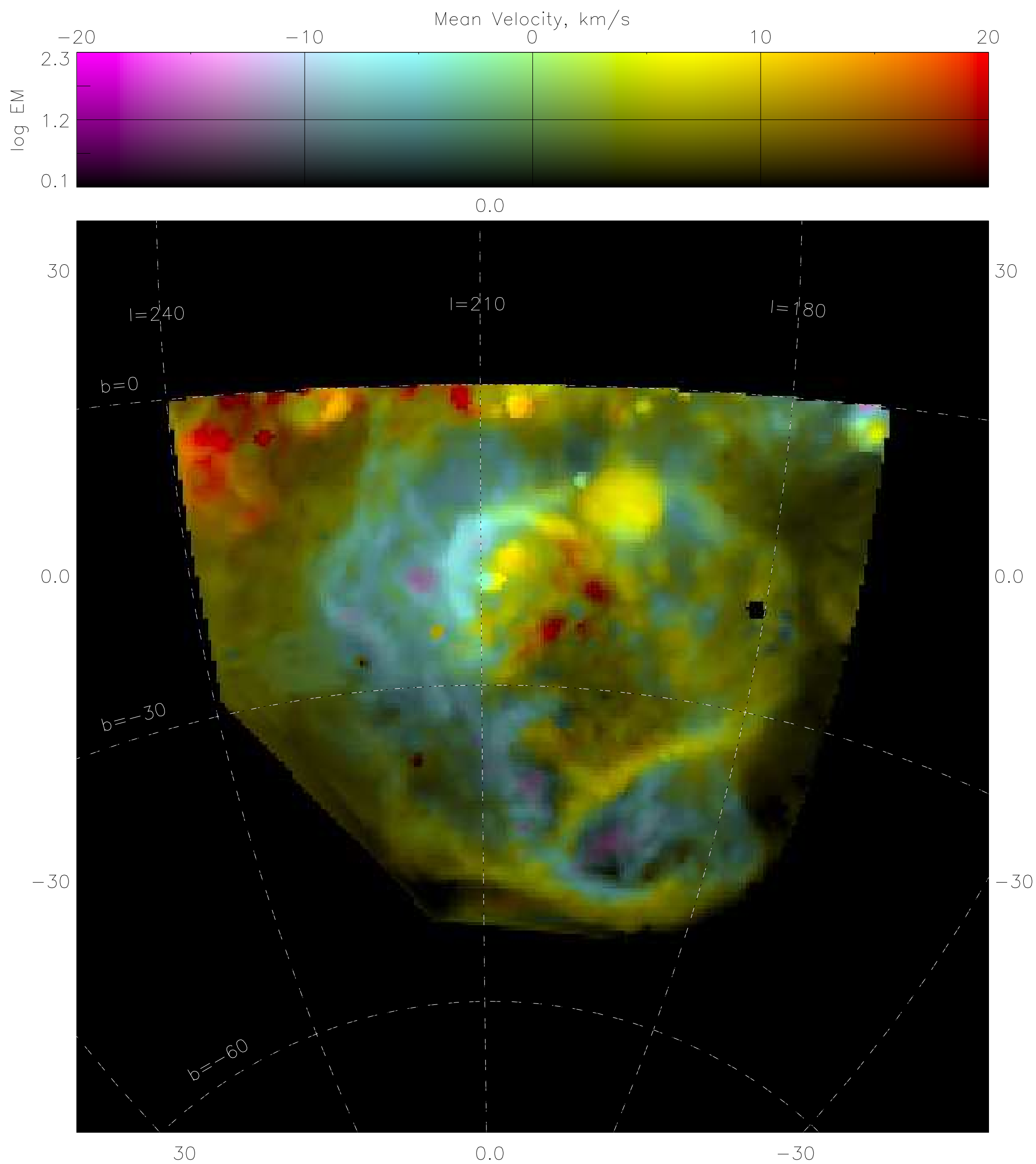

Fig. 1.— Example of a two-dimensional color image in which red, green, and blue represent different datasets. Obviously (we hope!), this image makes little sense with a black/white printer!

Here, the idea is to use color to represent the value of a single quantity, such as the mean velocity of a spectral line, and use the lightness (apparent brightness of the color: black is zero, full brightness is maximum lightness) to represent a different quantity, such as the integrated intensity of the spectral line. This is a *two-dimensional* mapping onto the image plane. Figure 1 shows an example. The image is Hα data from the WHAM survey; color represents mean velocity, lightness represents the logarithm of the line brightness.

The goal is to take two images, one of velocity and one of integrated brightness, and combine them into a single image in which color represents one image and intensity the other. Let's call these two images `velimg` (units are km/s) and `brtimg` (units are K-km/s).

3.1. 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. From the purist standpoint, you can obtain an ideal color table with the series of IDL commands

```
pseudo, 100, 100, 100, 100, 22.5, 0.68, colr
colr[*,0]= bytscl( colr[*,0], min=100,max=255)
r_orig = colr[*,0] & g_orig = colr[*,1] & b_orig=colr[*,2]
r_curr = r_orig & g_curr = g_orig & b_curr = b_orig
tvlct,r_orig,g_orig,b_orig
```

Here, the two sets of vectors `r_orig`... and `r_curr`... are what's actually used by IDL; `colr` is not used by IDL.

Better, however: Tim Robishaw and I have found that a slight revision to this purist concept (namely, removing a bit of red to make more green) gives an even better color table. You get it by replacing the above 5 statements with `pseudo_ch, colr`

This automatically writes `colr` into `r_orig`... and `r_curr`..., so you don't really need to include `colr` in the procedure call; it's provided only for your own edification, such as when you want to plot it as we've done in Figure 2.

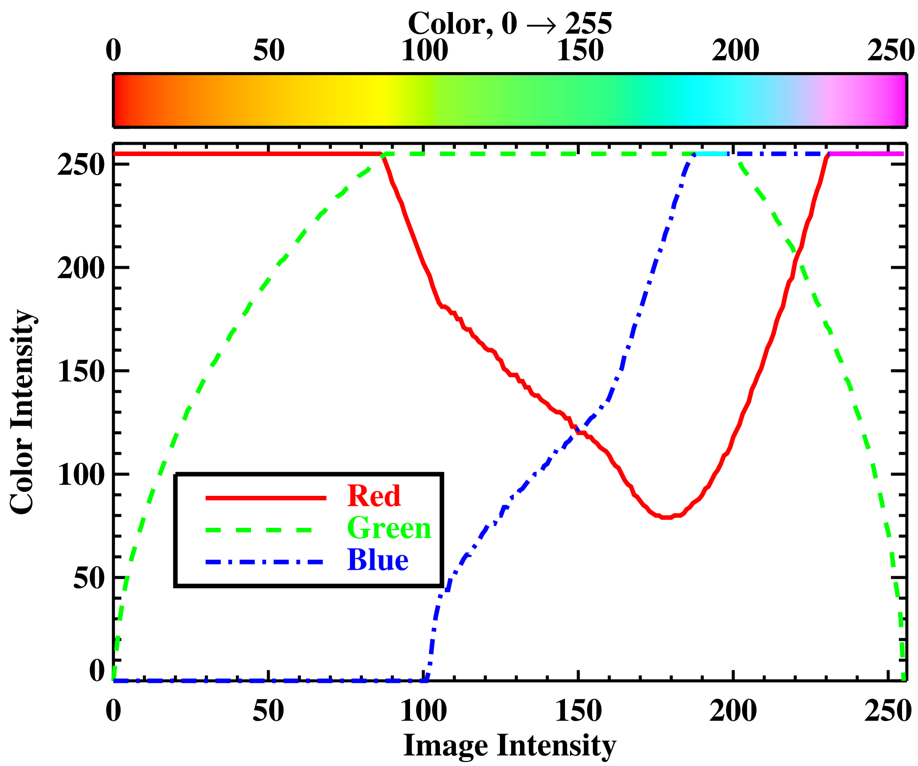

Fig. 2.— The color table created by *pseudo_ch*. The intensity of the three colors versus the input image value. This plot is best viewed in color! The IDL code to produce this plot is in the Appendix.

These **pseudo** procedures create a color table (here called `colr`) 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). Figure 2 plots the color table created by `pseudo_ch`. This color table runs from red to yellow to green to pale blue to magenta. 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 *colr[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.

3.2. Generating the color image

Having defined the color table, generating the color image is easy if you have the image of velocities; the velocities range between two extremes. The velocities of interest have a smaller extent, from `vmin` to `vmax`. You need to make this range correspond to the range of color indices, 0 to 255. You accomplish this with the `bytscl` function:

```
colimg= bytscl( velimg, min=vmin, max=vmax)
```

Note: a common mistake is to make `vmin` and `vmax` too far apart, so you're not gaining the full range of colors on the image. Be sure to experiment with these values!

If the line brightness were *everywhere the same, equal to its maximum value,* the three RGB images would be

```
redimg= colr[ colimg, 0]
grnimg= colr[ colimg, 1]
bluimg= colr[ colimg, 2]
```

Of course, the line brightness changes within the image, so you need to modify these three images to reflect that; we do that in §3.3, immediately below.

One little wrinkle: `colimg` is 2d while `(redimg, grnimg, bluimg)` will be 1d. You need to reform them back to 2d. For example, if the original dimensions of `colimg` are 480×240, then `redimg` will be a vector of length $480 \cdot 240 = 115200$. To convert it back to 2d, you'd enter

```
redimg= reform( redimg, 480, 240)
```

etc.

3.3. Changing the lightness for the second dimension

For any pixel, the *color* is given by the mixture of red, green, and blue specified by `(redimg, grnimg, bluimg)`. We add the line *brightness* information by intensity-scaling the three color images appropriately. The line-brightness array is `brtimg`; the brightness lies between two extremes. Suppose that we wish to display a narrower range within these extremes bounded by `bmin` and `bmax`, and that we want to apply a contrast power-law `gamma`. Then define the new integrated-intensity image as

```
newimg = ( ( ((brtimg - bmin)/(bmax - bmin)) > 0. < 1. )^gamma) ,
```

which is a float array lying between 0 and 1. We need to generate the three R,G,B byte (0 to 255) images by scaling the original, full-intensity `(redimg, grnimg, bluimg)` images all by the same factor for each pixel. We accomplish this by

```
r_img= byte ( (round( newimg * redimg ) > 0 < 255))
g_img= byte ( (round( newimg * grnimg ) > 0 < 255))
b_img= byte ( (round( newimg * bluimg ) > 0 < 255)) ,
```

The portions `> 0 < 255` are not necessary in principle, but we add them just to make sure roundoff error doesn't produce values outside the allowable range for bytes.

3.4. 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, r_img, channel=1
tv, g_img, channel=2
tv, b_img, channel=3
```

The other way, which works on both X and PostScript, is

```
tv, [[[r_img]], [[g_img]], [[b_img]]], /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 the third tutorial in our series, **MAKING, PRINTING, AND POWERPOINTING POSTSCRIPT FILES OF GRAPHS AND IMAGES**. Finally, if you want to annotate the images, you use the procedures in our second tutorial, **MAKING ANNOTATED IMAGES (WITH COLORBARS TOO!)**.

3.5. Experimenting with 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 perfectly in this two-dimensional color mode, you need to change the parameters and rewrite the image to the screen. An easy way to do it roughly is with `ct_fiddle`; it operates on all three colors identically, which isn't exactly what you want, but it's useful for experimentation. If you want to do it perfectly, then you need 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.

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

Here you want to represent three independent quantities by three independent, orthogonal-to-the-eye colors. Figure 3 shows an example. Red represents the Hα line intensity, green the 21-cm line intensity, and blue the XM band X-ray intensity.

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!

4.1. DirectColor and *ct_fiddle*: adjusting the image parameters

This is the ideal application for *DirectColor*. In DirectColor, each color has its own color table and you can manipulate them independently using Tim Robishaw's procedure `ct_fiddle`. 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.

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

In DirectColor, IDL takes control over all of the color tables. In other words, IDL forces a private color table. This means you'll get the "flashing" when the cursor moves on and off of IDL's display windows; see **BIDIDL**.

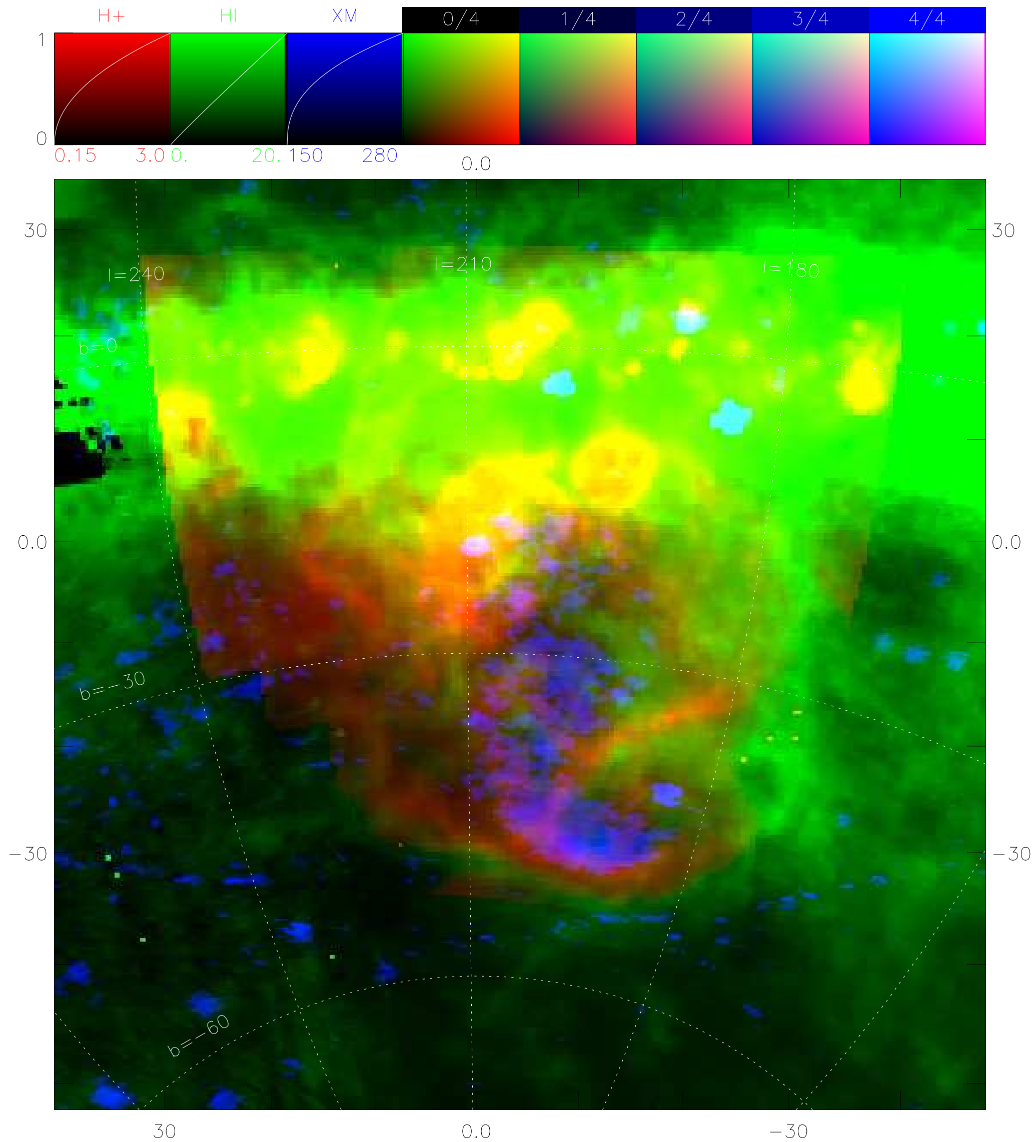

Fig. 3.— Example of a three-dimensional color image in which red, green, and blue represent different datasets. Obviously (we hope!), this image makes little sense with a black/white printer!

5. 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. For two-dimensional color, you need to also include a change in intensity in the orthogonal direction. The colorbar procedure discussed in our second tutorial, **MAKING ANNOTATED IMAGES (WITH COLORBARS TOO!)**, does this[1].

Three-dimensional color needs a more complicated colorbar: it has to display three dimensions on a two-dimensional area. We have attacked this with the scheme shown in our example Figure 3. 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. Try our not-well-documented procedure `rgbimg.pro`; if you need to, ask!

6. APPENDIX: THE IDL CODE FOR FIGURE 2

This is the batch file (invoked with '@') used to create the ps file for Figure 2. Note that `loadct,0` is called after calling `pseud0_ch`!!

```
;CREATE THE WINDOW WITH BORDERS...
w_left=0.16
w_rght=0.05
w_bot=0.16
w_top=0.14
space=0.03
width=0.1
img_cbar_posns, w_left, w_rght, w_bot, w_top, space, width, $
```

[1]There are other colorbar procedures. One common one is Fanning's. To our knowledge, ours is the only one that works for 2d color.

```
        imgposn, cbarposn, f_hor, f_ver

;CREATE AN XWINDOW IF PS IS 0...
if ps eq 0 then window,7,xs=f_hor*500, ys= f_ver*400

;CREATE THE COLORTABLE...
pseudo_ch,colr

;IMPORTANT NOTE: WHEN USING NONDECOMPOSED COLOR (PS OR X WITH
;PSEUDOCOLOR), YOU ***MUST*** INCLUDE THE FOLLOWING STATEMENT; AND
;INCLUDING IT WITH DECOMPOSED COLOR IS OK:
loadct,0

if ps then psopen, 'pseudoplot.ps', xsize=5, ysize=4, /inch, /color, $
  /times, /bold, /isolatin1

;MAKE NICE AXES AND PS FONTS USING SYSTEM VARIABLES...
!p.font=ps-1
!p.thick=6
!x.thick=4
!y.thick=4

;WRITE THE COLORBAR BEFORE CALLING SETCOLORS...
colorbar, pos=cbarposn, crange=[0,255], rgb=colr, color=!black, $
  xtit=textoidl('Color, 0 \rightarrow 255');;;;, ps=ps

;CALL SETCOLORS FOR THE VECTORIZED PORTION OF PS...
setcolors, /sys

;DO THE PLOT...
plot, colr[*,0], /nodata, position=imgposn, /noerase, $
        xra=[0,256], /xsty, xtit='Image Intensity', $
        yra=[0,260], /ysty, ytit='Color Intensity' , color=!black
oplot, colr[*,0], color=!red
oplot, colr[*,1], color=!green, lines=2
oplot, colr[*,2], color=!blue, lines=3

indxcyan=where( colr[*,1] eq 255 and colr[*,2] eq 255, countcyan)
if countcyan ne 0 then oplot, minmax(indxcyan), [255,255], color=!cyan
```

```
indxcyan=where( colr[*,0] eq 255 and colr[*,2] eq 255, countcyan)
if countcyan ne 0 then oplot, minmax(indxcyan), [255,255],
color=!magenta

legend, ['Red', 'Green', 'Blue'], lines=[0,2,3],
colors=[!red,!green,!blue], $
        position=[20,100], thick=[6,6,6], textcolors=[!red,!green,!blue]

if ps then psclose
setcolors, /sys

;RETURN EVERYTHING BACK TO NORMAL...
!p.font=-1
!p.thick=0
!x.thick=4
!y.thick=4
```

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