IDEAMAN—Image Display, Exploration, And MANipulation
April 16, 2016

Carl Heiles

Contents

1 PIXELS, COLORS, BYTES **2**

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

2.1 Combined Color: 256-Element Colortables 3

2.2 Decomposed Color 3

2.3 What Have I Done? 3

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

3.1 Linear Mapping, both Direct and Reversed 4

3.2 Nonlinear Mapping 5

3.3 Why We Need Nonlinear Mapping 5

4 DISPLAYING AN IMAGE IN IDL **6**

4.1 An Example: Read the Image from a FITS File 6

4.2 Displaying and Examining the Image with IDL's Native Procedures 7

4.2.1 Image Size and Window Size 7

4.2.2 Displaying the Image with `tv` 8

4.3 Manipulating the Image by Manually Changing the Colortable 8

5 REDISPLAY USING THE MODIFIED COLORTABLE **9**

6 EXPLORE AND HAVE SOME FUN WITH THIS IMAGE! **9**

This handout is oriented towards image exploration and manipulation—i.e., image processing—rather than making nice displays for presentation..

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.

---

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

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; 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. 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.

3.1. 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`.

3.2. 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`.

3.3. 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.

4. 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.

There are two ways to put the image on your computer screen (in computerese: "write the image onto your X window"). One 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.

4.1. 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/handouts_images.html`
or
`http://ugastro.berkeley.edu/radio/2016/index.html`
Copy this file to the disk area where you are running IDL. This file is in a format called "fits" format ("flexible image transport system"), 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 "mrdfits", which resides in the Goddard IDL library which, in turn, is already in your IDL path. If you are logged into `ugastro`, all you have to do is type

```
image = mrdfits( '/home/global/ay121/handouts/images/rass_c.fits', 0, hdr)
```

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.

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

Now we're ready to write the image to the X window. If all you want to do is look at it, the easy thing to do is use `display, image`. But when you want to examine the contents of individual pixels on the image, i.e., when you want to do image processing, you want each image pixel to occupy a single pixel on your screen.

4.2.1. 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 §4.2.2.

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)
```

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.

---

[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.

4.2.2. 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
```

4.3. Manipulating the Image by Manually Changing the Colortable

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". These images contain much more! To see more, we need to change the contrast—change the 'dynamic range'—by invoking a nonlinear mapping of data d to screen intensity I. In particular, 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 sample the image. You can get a quick feel for the interesting data range by just doing `plot, image` and visually estimating the range of interest. More instructive is to sample the image itself. Type

```
tr_profiles, byteimage
```

and follow the printed instructions; it plots profile cuts across the image, horizontally or vertically, at positions selected by the cursor. These profile plots are for the byte image. More useful is the *original* data array `image`:

```
tr_profiles, image
```

which plots the original numbers. 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.

After some inspection we see that limiting the data range to $0 \rightarrow 2000$ would indeed be a good start.

5. REDISPLAY USING THE MODIFIED COLORTABLE

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

and, instead of just the two strong sources, you see beautiful diffuse emission with lots of structure, which is produced by hot, $\sim 10^6$ K gas heated by the expanding shock of the supernova remnant.

6. EXPLORE AND HAVE SOME FUN WITH THIS IMAGE!

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. For detailed pixel-by-pixel sampling, try `rdpix, image`. 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`; this is a simple way to see most of the interesting structure:

```
tv, hist_equal(image)
```

You can rotate with `rotate` or `rot`, `transpose`, `zoom`, draw `contour`s, label your images and make

coordinates using `plot` (with the `/noerase` keyword) and `xyouts`, etc., etc., etc.

Play around with contrast by experimenting with d_{bot}, d_{top}, and especially `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.

To learn the range of native IDL capabilities, enter IDL and type "?", which brings up the online help window. Go to the bottom left and click on "Contents", then "Routines (by topic)", and take a look at the sections entitled "Array Manipulation", "Color Table Manipulation", "Direct Graphics", "Image Processing", "Plotting", "Signal Processing".