BIDIDL: BASIC IMAGE DISPLAY IN IDL: IMAGES WITH A SINGLE 256-ENTRY COLOR TABLE
January 14, 2007

Carl Heiles

Contents

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

1. VISUAL CLASSES AND YOUR IDL START-UP FILE

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); or 1024 × 768 (4 × 3 aspect ratio)—found on older computers and small laptops (the 4 × 3 format comes in much larger dimensions, too, like 1600 × 1200 pixels). Typical CCD projectors (as used for PowerPoint) all have the 4 × 3 format with 1024 × 768 pixels as their native resolution, so when you use a small laptop with such a projector you have the best match. Each pixel is small! But if you look carefully, you can see them.

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". However, some computer screens can't display this full range of colors. Instead, they can display only 256 different combinations. This may seem small, but for most purposes it's exactly what you need. For example, in a black/white image there can be 256 (that's 8 bits worth) different intensity levels, ranging from black, through the grays, to the brightest white.

Our computer screens fall into several categories called *visual classes*. One important aspect of the visual class is the overall type of color table it supports, either: *dynamic*, which enables cursor control of image contrast and minimum/maximum intensities; and *static*, which does not offer that control. A dynamic color table is usually what you want to manipulate images, very useful in astronomy! See §3. What you get depends on the capabilities of your video card and software driver, which in turn depends on the type of computer and operating system. In order of increasing sophistication, these categories include:

1. *GrayScale*, with 256 shades of gray and a dynamic color table that allows only differing intensities of white/black (i.e., gray). Found only on Sun machines.

2. *PseudoColor*, with 256 colors and a dynamic color table (see below). Found only on Sun machines. PseudoColor is usually what you want to manipulate images because it allows you to use color instead of gray and to use the cursor to change the contrast.

3. *TrueColor*, with 256^3 colors and three static color tables, one for each color. Found on nearly all Sun machines, on all PCs running windows, also on PCs running Linux, and on Mac OSX. This gives you millions of colors but no cursor manipulation of image contrast.

4. *DirectColor*, with 256^3 colors and three dynamic color tables, one for each primary color. Found on those few Sun machines that have "Creator 3D" graphics and, also, on PCs running Linux. This is the most flexible mode and provides all of the above capabilities.

5. *Decomposed Color: Important point about TrueColor and DirectColor.* As you will learn below, often you want a 256-entry color table. This is the equivalent of PseudoColor. In

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

contrast, the normal operation of TrueColor and DirectColor has 256^3 colors. This is known as *decomposed color.*

You can make TrueColor and DirectColor use a 256-entry color table by turning off decomposed color, done with the IDL command `device, decomposed=0`. You can turn it back on with (guess what!) `device, decomposed=1`. Of course, in TrueColor this pseudo-PseudoColor table will be static, not dynamic as it is in DirectColor.

The remainder of this tutorial assumes you have turned color decomposition off, i.e. you are using a 256-entry color table.

2. TELL YOUR MACHINE WHAT YOU WANT TO DO!

Unless you have a restricted machine like PC/Windows, Mac OSX, or a cheap Sun (machines for which you have no choice!), you have to tell IDL what color mode you want. In IDL, you must choose the mode first thing. With our locally-preferred startup file you are queried for this choice. Otherwise, to get pseudo (available only on Suns): after entering IDL...

device, pseudo=8

(The "8" stands for 8 bits; remember, $2^8 = 256$). To get true or direct color, it's...

device, true_color=24 , retain=2

device, direct_color=24 , retain=2

The **retain=2** means that IDL will refresh the window if it gets covered up by another window. You should use this option on Linux, some versions of which don't provide this service.

If you request a color mode that your computer doesn't support, IDL will tell you so and revert to the most capable mode that it does support. Once you're in IDL, you can check to see what you've told IDL. Enter the IDL command...

help, /device

and IDL will tell you what display device it is using. If you are in PseudoColor, the response will show, among other things: Display Depth, Size: 8 bits, (xxxx, yyyy), where xxxx and yyyy are the screen pixel dimensions; Visual Class: PseudoColor (3); Colormap: Shared, zzz colors. It also lists all the windows you have open. 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. 256-ENTRY (8-BIT) COLOR TABLES

To follow the writeup below on your computer, get into PseudoColor, or DirectColor with color decomposition turned *off* so that you have a 256-entry color table. DirectColor with decomposition off mimics PseudoColor because, by default, its three RGB color tables are set equal.

In 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 making the paper black uses lots of printer toner, which is expensive. The *third* reason is that in scientific journals, images with the reversed mapping are reproduced much better. 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.

This particular nonlinear mapping can be invoked easily and automatically in IDL by typing **xloadct** or—better for interactive work on Linux platforms—Tim Robishaw's `ct_fiddle`. (It's better because the image actually changes as you move the cursor). These programs allow you to manipulate the color table with the mouse: you can change the values of γ, d_{bot}, d_{top} smoothly and watch the contrast of your image change. **xloadct** also allows you to select a reversed mapping and to select a multitude of predefined color tables, not only grayscale but many others. Once you get an image, it's fun to experiment. **xloadct** also allows you to generate any completely arbitrary nonlinear mapping; click on "Function" and experiment. Alternatively, you can load predefined color tables; one of my favorites (but undesirable from a purist's standpoint—see our memo *Color Images to Represent One, Two, and Three Dimensional Data*) is "STD GAMMA II", which you can load from `xloadct` or manually using **loadct, 5**.

There is one other commonly used nonlinear mapping, the so-called "histogram equalization" technique. In this technique, the mapping is modified 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. LET'S TRY IT IN IDL!

First, generate an image. There's a nice image of the X-ray sky, obtained by the ROSAT satellite, in `~heiles/courses/handouts/images/rass_c.fits`; or on the web at: `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

$$\mathbf{image = readfits('\sim heiles/courses/handouts/images/rass\_c.fits', headerinfo)} \tag{4}$$

This returns two arrays: the image array (**image**) and information about the image (**headerinfo**); type **hprint, headerinfo** to see the header information. Now type **help, image** and IDL will tell you that it is a 480×240 FLOAT array. 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. Never mind! Display the image anyway by typing

$$\mathbf{tv, image} \tag{5}$$

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 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**...

$$\mathbf{byteimage = bytscl(image)} \tag{6}$$

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

$$\mathbf{tv, byteimage} \tag{7}$$

A quick (but slightly inaccurate) alternative to the above is IDL's **tvscl**, which combines the two operations.

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, type **xloadct** and manipulate the sliders, or use `ct_fiddle` . 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 $0 \rightarrow 4$. This provides very little dynamic range for this glow, so we need to expand this range so that we can see its structure more clearly.

How much should we expand the range? We might make a guess and try $-174 \rightarrow 2000$. If that didn't give a nice result, we could try some other values. But we don't have to guess! IDL provides a nice way to interactively print the values of the image. We use **rdpix**, which prints out the pixel values of an image that one must specify as we 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...

$$\mathbf{rdpix, image} \tag{8}$$

and then the printed numbers will be of the *original* data array **image**. From this we see that limiting the data range to $0 \rightarrow 2000$ would indeed be a good start.

You can also get a quick feel for the interesting data range by just doing **plot, image** and visually estimating the range of interest.

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

$$\mathbf{byteimage = bytscl(image, min = 0, max = 2000)}\ . \tag{9}$$

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 use **xloadct** to play around with contrast—looks great, eh? 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.

Nicer than **rdpix** is the native IDL procedure **profiles**, which makes a plot of the pixel values along either a horizontal or vertical line, depending on what mouse buttons you push. Nicer still is Tim Robishaw's enhanced version, `tr_profiles`. This procedure can show the values of either the data array you are actually seeing (**byteimage** in this case) or any other data array of the same dimensions (**image** in this case). So try

$$\textbf{profiles, image} \quad \text{or} \quad \textbf{tr\_profiles, image} \tag{10}$$

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. 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,

$$\textbf{profiles}, (\mathbf{0} > (\textbf{image} < \mathbf{5000})) \tag{11}$$

Here, the ($\textbf{image} < \mathbf{5000}$) means "take whichever number is smaller, either the data value in **image** or the number 2000"; and the $\mathbf{0} > \mathbf{X}$ means "take whichever number is larger, either the data value in **X** or the number 0".

Often even better is to 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* by typing. . .

$$\textbf{indices} = \textbf{defroi}(\textbf{xsize}, \textbf{ysize}) \tag{12}$$

where **(xsize, ysize)** is the size of **image**; then **image[indices]** is an array containing only those image pixels within the area you've selected—try **histo = histogram(image[indices])**.

Note: we have two procedures like `defroi` that work by selecting points in graphs. One procedure, `graphselect`, finds points inside a closed area that you define with the cursor. The other, `pointsinside`, finds points inside a closed area that you have plotted using IDL commands.

4.1. atv

atv is a package you might find desirable for some purposes. It offers a widget-based driver for many of the basic image display functions. It's on your path if you use our startup file. Also, it can be downloaded here: `http://www.physics.uci.edu/~barth/atv/`

5. IMAGE SIZE AND WINDOW SIZE

This image that we've been playing with is a nice, convenient size for viewing. But it probably doesn't fill the window area, or maybe the window is too small. 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

$$\mathbf{window}, \mathbf{xsize} = \mathbf{480}, \mathbf{ysize} = \mathbf{240} \tag{13}$$

and then redisplaying the image with **tv, byteimage**. Or, suppose you want to make this image *larger* so that you can see more details. You could do this with **zoom**, but if you want to make the whole image larger you need to increase 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 3.5 in the vertical directions, i.e. to make an array of size 960×840. Do this by

$$\mathbf{bigimage} = \mathbf{congrid}(\mathbf{byteimage}, \mathbf{960}, \mathbf{840}) \tag{14}$$

Then create an appropriately-sized window (e.g. with **window, 5, xsize=960, ysize=840**—which creates a new window, numbered 5, and leaves the old ones in place), use **tv, bigimage**, and...there you are. There's another routine called **rebin**, which works only for integral factors.

CAUTION WITH **congrid** *and* **rebin***:* These routines 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.

6. ANNOTATING IMAGES; MAKING POSTSCRIPT COPIES OF IMAGES

Go check out our other tutorials in this series! See **FOUR TUTORIALS ON IMAGES: NOVICE → EXPERT**.

7. FLASHING

Sometimes the whole terminal screen changes colors when you move the cursor onto an IDL window. This is commonly known as *flashing* and it's a royal pain. Flashing can occur when using dynamic color tables. It always occurs in PseudoColor with 256 colors and it always occurs in DirectColor. It sometimes occurs in PseudoColor *without* 256 colors.

The video card has a single set of color tables that apply to the whole screen. Its color tables can be shared among different applications (*shared colormaps*) or totally grabbed by single applications (*private colormaps*). The operating system uses colors for its displays, generating them in a shared colormap. Similarly, you generate your desired color tables within IDL. Obviously, the system and your IDL color tables are different.

For IDL running DirectColor and 256-color PseudoColor, IDL's color tables are private—they have to be, because they completely fill the video card's color table definitions. Under these conditions, when the cursor is not on an IDL graphics window the operating system has control of the video card's color tables, and the system colors are displayed. When you move your cursor to an IDL graphics window, the IDL color tables rule. So you get flashing. The flashing also affects the operation of `xloadct`. `ct_fiddle` doesn't have this problem.

In contrast, with ordinary PseudoColor, IDL uses a shared colormap and grabs only the colors that are unused by the system. Ordinarily, Solaris uses perhaps 30 colors for its own sake. This leaves perhaps 226 colors; IDL grabs these for its own—but the colormap is shared, so you *don't* get flashing.

However, if other applications—in particular, Netscape or Mozilla—are running before you start IDL, then they grab their own sets of colors from the shared colormap, leaving fewer colors for IDL. Under these circumstances, you might have very few colors available for IDL itself. Alternatively, if you start Netscape after starting IDL there are very few (maybe none!) colors available for Netscape, and its displays will look weird.

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.

8. MORE, MORE, MORE…

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. You can play with color tables with **xloadct** and `ct_fiddle`. You can do "histogram equalization" with **hist_equal**. 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. In the IDL Handiguide, look under "Array Manipulation", "Array and Image Processing", "Window Routines", "Direct Graphics Plotting Routines, Two-Dimensional".

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