BIDIDL: BASIC IMAGE DISPLAY IN IDL
March 18, 2003

1. COLOR DISPLAYS 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 are 1280×1024 (or, if your research advisor is cheap, 1280×768). So each pixel is small! But if you look carefully, you can see them—in particular, on a field of uniform color you can see thin vertical lines, and these mark the vertical pixel boundaries.

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 greys, to the brightest white.

Our computer screens fall into several categories. These depend on the capabilities of your video card and software driver.

1. *Pseudocolor*, with 256 colors and a color table (see below). Found mainly on older Sun machines and also on current cheap Sun machines; if you have this you are in some sense lucky. This mode allows you to use the cursor to change the contrast of images.
2. *Truecolor*, with 256^3 colors and no color table. Found on current cheap Sun machines, on all PC's running windows, also on PC's running linux. This gives you millions of colors but no cursor manipulation of image contrast. *Note:* Current cheap Sun machines can run either pseudocolor or truecolor, but Kelley has to make the change—meaning you have to choose one and stick with it.
3. *Directcolor*, with 256^3 colors and three color tables, one for each primary color. Found on Sun machines with "Creator 3D" graphics and, also, on PC's running Linux. This gives all capabilities.

Portions of the first portion of this writeup deal with using the cursor to control contrast,

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

for which you need either pseudocolor or directcolor. If you have only truecolor, some of the discussion won't apply to you, but forge ahead anyway.

IMPORTANT: You have to tell IDL what color mode you want. To follow the writeup below, try to get into pseudocolor; you do this by making the *first* statement you type in after entering IDL...

device, pseudo=8

(The "8" stands for 8 bits; remember, $2^8 = 256$). It's best to put this statement in your IDL startup file. This won't work if you have a PC under Linux, so instead try

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 don't need to do this on the Sun machines because Solaris takes care of this detail.

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 (zzz will probably be about 228; see §7). 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.

2. COLOR TABLES

In pseudocolor mode, the maximum number of color/intensity combinations that can be displayed simultaneously is 256.[2] 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 "grey-scale"

[2] Actually, it's less—probably 228. See §6.

images. Grey, or white, is composed of an equal mixture of red, green, and blue, and all we deal with is the intensity I. In a grey-scale 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 grey-scale image, the mapping between data value and intensity.

2.1. 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 colortable, see §5.

2.2. 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, alternatively, by using our homegrown **diddle**. **diddle** works even on directcolor, while **xloadct** words only on pseudocolor. These programs allow you to manipulate 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 grey-scale 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 is "STD GAMMA II", which you can load 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**.

3. LET'S TRY IT IN IDL!

First, generate an image. There's a nice image of the X-ray sky, obtained by the German ROSAT satellite, in */dzd2/heiles/courses/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 aleady in your IDL path. All you have to do is type

image = readfits('/dzd2/heiles/courses/handouts/rass_c.fits', headerinfo) (4a)

on the *astron* cluster, or

image = readfits('/home/ay120b/idl/rass_c.fits', headerinfo) (4b)

on the *ugastro* cluster.

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. 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 grey 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 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 **diddle** . 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 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...

$$\mathbf{byteimage = bytscl(0 > (image < 2000))} \tag{9}$$

Here, the (**image** $<$ **2000**) 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 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 a 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 **profiles**, which makes a plot of the pixel values along either a horizontal or vertical line, depending on what mouse buttons you push. 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

$$\mathbf{profiles, image} \tag{10}$$

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, as above; for example,

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

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...

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

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])**.

4. 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. 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.

5. MAKING HARD COPIES OF IMAGES

OK...you've got a beautiful image and you want to make a hard copy on the printer. Some considerations:

- The screen usually shows white on black. As we mentioned above in §2, This doesn't come out very well on the printer; black on white works better. Also, we use a *lot* less printer toner with black on white. You can get black on white with **xloadct**: click on "Options" and choose "Reverse Table". You'll have to play with the Gamma Correction and Stretch sliders again to get what you want.

  Alternatively, you can type

  $$\mathbf{tv}, \mathbf{not}(\mathbf{image}) \tag{15}$$

  because the **not** operating on a byte array reverses the bits and maps $0 \rightarrow 255$ into $255 \rightarrow 0$.

- How big do you want the printed image?

5.1. Copying the screen to PS with hardimage

The simplest and least elegant way to get a ps version is to copy the screen pixels directly onto postscript. It's least elegant because any notations you make get transferred to PS as pixellated chatacters, which aren't very pretty.

You might be tempted to use **hardplot**. But that's written mainly for line plots, which usually don't have shades of grey. For images use **hardimage**. Using its keywords, you can generate a postscript (PS) or encapsulated postscript (EPS) file, make an image of size that you specify, and other options. They you examine the file using the unix command *xv* or *display*, and print the file using the UNIX command *lp filename*. Note: **hardimage** gives you an *exact copy* of what you see on the screen: to get the desired black-on-white image you need from the printer, you need to create this same version on the screen. **hardimage** also does color, and it works in all three color modes.

5.2. Generating PS directly with openimageps

It's much better to write the image itself directly onto PS instead of first onto X, to avoid pixellation with character notation. To accomplish this, make PS the output device; do all of you **tv**'ing and character annotation; and then close the PS device.

To make PS the output device, you need to create a PS window with the **set_plot, ps** and **device** commands. In the **device** command, you specify such things as size of the window on the printed page; it will look something like this:

```
device, filename=filenm, bits=8, landscape=landsc, /inch, /color, $
xsize=xinch, xoff=xoffset, ysize=yinch, yoff=yoffset
```

where **xsize** is the size of the PS window (not the image) and **xoffset** is the horizontal offset of the lower left-hand corner of the PS window from the corner of the page. The units here in inches because of the **inch**. There's also a **landscape** option; if you use it, look carefully at the documentation!

You can save yourself a lot of grief by using our **openimageps** procedure, which does all this for you, and you can look at that code to see exactly how it's done. **openimageps** works in any color mode.

After opening the PS window, do all the steps required to generate the image and its notation; then finish with **closeps**

6. ANNOTATING IMAGES WITH PLOT BORDERS AND LABELS

Most of the time you will need to annotate an image with a plot border or other annotations. Here's a verbal description of the basics, in which we assume the image is the 480×240 pixel image from */dzd2/heiles/courses/handouts/images/rass_c.fits*.

The details depend on whether the output device is X windows (X) or Postscript (PS). More precisely, what works for PS *also* works for X, so if you have the remotest idea that you'll want to make a PS file then you should use the more general notation.

6.1. X-windows **ONLY**

First, you need to make a large enough window to leave room for the plot border and labels. You can do this by opening a window that's larger than the image size. In the example, we make the window 50 pixels larger all around than the image.

You need to center the image within this window. Do this with the x,y inputs to the **TV** command:

```
TV, image, 50, 50
```

Next, you need to create a plot border that's aligned with the image. For this you use the **position** keyword in the **plot** command. Also, you don't want the plot command to erase the image, so you use the **noerase** keyword; and also you probably want to create the plot border without plotting data, so you use the **nodata** keyword. With all this, the **plot** command can look like this:

```
plot, [0,0], [0,0], xra=[360,0],yra=[-90,90], xsty=1, ysty=1, $
/nodata, /noerase, position=[50,0,529,339],/device, $
xtit='GALACTIC LONGITUDE', ytit='GALACTIC LATITUDE'
```

Here the **device** keyword specifies that the position coordinates are given in device units, which are pixels for the X window.

6.2. X **AND** PS

The main difference in PS is that you cannot specify things in **device** coordinates; rather, you must use **normalized** coordinates. In X, you can use either. Normalized coordinates run from $0 \rightarrow 1$ and are the fraction of the total image size in that particular direction. The necessity for normalized coordinates arises because PS uses scalable pixels—meaning that pixels are not necessarily square because the pixel shape depends on the ratio of X and Y sizes of the image that you specify in the **TV** command to the number of X and Y pixels in the image.

You can use normalized coordinates with X, too. This means that you can use *exactly* the same code for image display for both X and PS, with the only exception being that you open a PS window or an X window.

To begin with, you need to position the image within the window (as opposed to the **device** command, which positions the PS window on the page). In our example, you might wish to center it. For PS (but not for X) You need to specify the image size. The following works in both:

```
xoffset = 50/580.
yoffset = 50/340.
xplotsize=480/580.
yplotsize=240/340.
tv, image, xoffset, yoffset, xsize=xplotsize, ysize=yplotsize, /normal
```

This works for both because, under X, **TV** ignores the **xsize** and **ysize** keywords (because, under X, the image size is always in pixels, equal to the size of the array *image*).

Now for the plot border. This is just like the description above for X, except that you need to use normalized coordinates. So you'd have

```
plot,x,y,xra=[360,0],yra=[-90,90], xsty=1, ysty=1,/nodata, /noerase, $
        position=[50/580., 50/340., 529/580., 289/340.],/norm, $
        xtit='GALACTIC LONGITUDE', ytit='GALACTIC LATITUDE'
```

If you are attentive, then you will have asked: "Why divide by 580 instead of 579? Why divide by 340 instead of 339?" *Answer*: "I don't know the answer, but that's what works!"

If you're lazy, or clever, then you don't need to calculate the normalized coordinates yourself. Instead, you can use IDL's **convert_coord** function.

Of course, you can annotate with **xyouts**; you just have to remember to use either **norm** or **device** coordinates!

6.3. ATV

For one-d images, **ATV** is a package you might find desireable for some purposes. It offers a widget-based driver for many of the basic image display functions. It's located at */deep2/dfink/cvs/idlutils* .

7. 256 PSEUDOCOLORS? THIS IS A DETAIL THAT YOU CAN IGNORE UNLESS YOU WANT AN EXCELLENT POSTSCRIPT VERSION...

Above, in §1 and §2, we discussed how your screen, in pseudocolor mode, can show 256 intensities (or, more generally, color/intensity combinations). That is absolutely correct!

But IDL doesn't have all 256 combinations available to it because the operating system takes up some colors for itself[3]. It usually uses 28 colors for things like the background screen color, window color, window borders, typescript color, etc. This leaves (usually) 228 color/intensity combinations for IDL[4]. IDL takes care of this little detail automatically, so you don't have to worry about it. But this difference is frustratingly annoying when you are trying to make a high-quality postscript image.

For the true aficionado, you can get around this and force IDL to use all of the available 256 color/intensity combinations. If you do this, then the screen colors will "flash" depending on whether your cursor is on an IDL window or not—and also **xloadct** won't work the way you'd like (but **diddle** will!). If you really want to invoke this option—and we don't usually recommend

[3]see *Quick IDL Tutorial: Color Images, 8-bit and 24-bit* for a more complete discussion.

[4]When you type **help, /device**, the number available to IDL is listed under "Colormap": "Shared" means IDL is sharing the colormap with the system, and then it gives the number of colors available to IDL.

it except when you're making postscript files—then when you enter IDL, after typing **device, pseudo=8**, immediately create the first window by typing **window, colors=256**. That forces IDL to use 256 colors for the whole session. With this, when you type type **help, /device**, the "Colormap" listing says "Private, 256 colors.": IDL overrides the system's color allocations. Private colormaps always cause flashing.

Directcolor *always* has 256 intensities in each color—i.e. it always uses a private colormap. This means that you always have flashing. In contrast, truecolor cannot use a colormap, so there can be no flashing.

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 **diddle**. 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".

Most importantly: if you are using color for display purposes, then read our handout *Color Images to Represent One, Two, and Three Dimensional Data.*