1d IMAGE EXPLORATION AND PROCESSING USING `display`
April 17, 2017

Carl Heiles

Contents

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; the somewhat larger 1440×900 (8×5 aspect ratio); 1024×768 (4×3 aspect ratio)—found on older computers and small laptops. These aspect ratios come in much larger dimensions. 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 desirable 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 PseudoColor, which is 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 and then defining a colortable. You do the former with the command

```
device, decomposed=0
```

By default, IDL sets the three colors `r,b,b` equal, which gives you grayscale. If you want the system colors (like `!red`) defined and you're using our IDL library, the uppermost 12 entries of the grayscale colortable define the colors[2], and you load that colortable with the command

```
setcolors, /system_variables  (or setcolors, /sys )
```

Alternatively, you could define a modified grayscale (e.g., as in equation 3) or a false-color colortable that is not gray[3].

2.2. Decomposed Color

For ordinary non-image work you almost always want Decomposed Color 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.
```

If you want the system colors (like `!red`) defined and you're using our IDL library, also type the

---

[2]e.g., number 244 is `!black`, 245 is `!red`, etc.; use IDL's `tvlct` comomand with the `get` keyword set to see these definitions.

[3]IDL has a wide selection of such colortables; in IDL, to view them type `xloadct` and to load one use `loadct`.

command `setcolors, /system_variables  (or setcolors, /sys )`

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, because images in PostScript need decomposed color, while vector graphics (like symbols or lines) need combined color. 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. 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[4]. 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.*

[4]You do this with `device, decomposed=0` and, if desired, `setcolors,/sys`.

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 USING `display`

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.

The best way to display your image is with Tim Robishaw's (TR's) `display`. It's quick, simple, and elegant, and we recommend it for most purposes including exploration and producing publication-quality images. We'll describe using it on a nice image of the X-ray sky `rass_c.fits`, obtained from the course website under Lab 4. To read the data file into an array called `image`:

```
image = mrdfits( 'rass_c.fits', 0, hdr)
```

This returns two arrays: the image array (`image`) and information about the image (`hdr`); type `print, hdr` to see the header information. Now type `help, image` and IDL will tell you that it is a 480×240 FLOAT array. You an display this image on any size window, even a tiny one, but to be able to see each pixel individually the window must be at least this large.

To display this image:

```
display, image, out=out
```

(Note: `display` has many options and totally flexible; you can make the displayed image arbitrarily fancy. See the documentation.) The keyword `out` is a structure containing some basic information used by `trp` (see below).

5. EXPLORING THE IMAGE WITH `trc` AND `trp`

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". Where are these sources located? Use `trc` to find the pixel coordinates of these sources. `trc` is just a wrapper that calls `tr_rdplot` with useful settings. Type

```
trc, x, y
```

Then move the cursor inside the plot window. Left clicks write and record the `(x,y)` values in the coordinate system you select (i.e., `data, device, normal`; default is `data`). A middle or left click will exit the program. Notice that the crosshairs are clipped at the axes: it's often useful to make the crosshairs extend to the edges of the window. This is accomplished by passing the `/noclip` keyword, and since we can use minimum matching when calling keywords in IDL, `/noc` will be good enough:

```
trc, x, y, /noc
```

If you don't want to store the positions of your left-clicks in `x` and `y`, then you don't need to include them in the procedure call. For more details, see the documentation of `trc`, `trp`, and `tr_rdplot`.

These images contain much more than just these two sources! There's lots of low-level diffuse emission that we can see only by increasing the visibility of weak emission. We need to 'clip' the point-source peaks to limit their minimum and maximum values using the keywords `min` and `max`. We could guess some trial values, but 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 using `trp`, which plots horizontal or vertical cuts of the image. Type

```
trp, out
```

and follow the printed instructions; it plots profile cuts across the image, horizontally or vertically, at positions selected by the cursor. Move the cursor inside the plot window containing the displayed image. You now see the profiles in the newly created window (labeled *Profiles*). Initially, you see intensity versus the `x` data coordinates. Left click toggles betwwen horizontal/vertical; right click exits the program and closes the profiles window. From all this, a reasonable clip range is 200 to

1400, so we redisplay the image:

```
display, image, min=200, max=1400, out=out
```

This looks good, but the weak parts are still hard to see. They can be enhanced by increasing the contrast, using gamma less than one. Example:

```
gamma = 0.25
display, image^gamma, min=200^gamma, max=1400^gamma, out=out}
```

You can play with `min, max, gamma` to achieve nirvana; 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. 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` ("defiine region of interest"). You can do "histogram equalization" with `hist_equal`; this is a simple way to see most of the interesting structure:

```
display, hist_equal(image)
```

You can rotate with `rotate` or `rot`, `transpose`, `zoom`, draw `contours`, 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".