IACIDL—IMAGE ANNOTATION AND COLORARS IN IDL: —SOME LOWER-LEVEL DETAILS FOR SPECIALIZED APPLICATIONS— August 27, 2011

Carl Heiles and Tim Robishaw

Contents

It's easy to create labelled and annotated images using `gdisplay`. But sometimes you need to use lower-level software for specialized apploications. Suppose you have a single two-d array and want to make an image (that's one-dimensional color). The x and y axes have meaningful values and you want the image to incorporate those on a plot scale. And perhaps you want to make a contour plot that overlies the image. Here's a quick tutorial.

1. THE GLOBAL VIEW

1.1. The Order in Which We Do Things

When doing PostScript, the color table for the annotating characters can be different from that for the image. It's easier to change these colortables only once. Moreover, you might as well adopt this philosophy for both X and PostScript. For this reason, when you include a colorbar, it's much easier to first write both images and then write their annotation. This explains why we order the actions as we do. The actions consist of:

1. Produce a window having a nice layout of borders (§2).
2. Write the data image to the window with its particular colortable (§3).
3. Write the colorbar image to the window using the same colortable (§4).
4. Change the colortable to write characters (§5.1).
5. Write annotation for the data image (§5.2).
6. Write the annotation for the colorbar image (§5.3).

If you want, you can interchange the order of doing the data and colorbar. The following sections describe these actions in detail.

1.2. Our Version of `colorbar`

One more thing: if you want a colorbar, you can use *Robishaw's* `colorbar` as illustrated here; you get it automatically if you use our recommended IDL startup file. There's another colorbar procedure out there from Fanning; it's different and if you use it you are on your own (you can check by typing `doc_library, 'colorbar'`).

1.3. The variable named `ps`

In keeping with our handout PSIDL on making PostScript, we define the variable `ps`: equal to 0 for writing to the screen and 1 to PostScript. For PostScript we use hardware fonts and for X we use vector fonts, so when writing characters we include the parameter `font=ps-1`. Our IDL statements below include these parameters.

2. PRODUCE A WINDOW HAVING IMAGE BORDERS AND ROOM FOR A COLORBAR

You are writing either to an X window or PostScript file; we refer to either as the "window". In either case, you need to make the window larger than the image so that there's room for the annotation and/or the colorbar. You usually don't want equal margins on all sides of an image. This wastes some space because you need to provide margin space for coordinate labels, which usually appear on only the left and bottom, so these two margins need to be wider than the right and top. Also, you might wish to include a colorbar, usually on on the top or the right-hand side, and that complicates the calculation of margins.

So your first task is to think about aesthetics and practicality and decide how much space you want or need for the borders. You then use IDL to *implement* your decision. Here we make the implementation easier and we also provide a straightforward way to generate a colorbar in §4.

2.1. Define Those Margins

Our procedure `img_cbar_posns` calculates positional information given your specification of margin widths. Here, the word "margin" means the white space surrounding the image and, also, the colorbar if there is one. The inputs are:

1. `w_left`, the distance from the left-hand edge of the image to the left-hand edge of the plotting device, *in units of the width of the image.* This is the width of the left-hand margin, within which labels will go and, also, some blank space around the image. A reasonable value is 0.1, which leaves some room for the vertical axis labelling.

2. `w_right`, the width of the right-hand margin. Often you don't need room for labelling on the right, so you could use a small value like 0.05.
3. `w_bot`, the width of the bottom margin in units of the height of the image. A reasonable value is 0.1, which leaves some room for the vertical axis labelling.
4. `w_top`, the width of the top margin. Often you don't need room for labelling on the top, so you could use a small value like 0.05.
5. `space`, the distance between the image and the colorbar in units of the image size (vertical size for a horizontal colorbar on top, horizontal size for a vertical colorbar on right hand side). If there is no labelling on top of the image, this can be small (e.g. 0.03). If you don't have a colorbar, set `space=0`
6. `width`, the width of the colorbar in units of the image size, as with `space`. A value of 0.1 often looks good. If you don't have a colorbar, set `width=0`

There are four outputs, which are used as the inputs to other procedures. The outputs are:

1. `imgposn`, the 4-element vector for image corner positions, normalized coordinates
2. `cbarposn`, the 4-element vector for colorbar corner positions, normalized coordinates
3. `f_hor`, the horizontal window size in units of the horizontal image size
4. `f_ver`, the vertical window size in units of the vertical image size

For no colorbar, the procedure call would be

```
img_cbar_posns, 0.1, 0.05, 0.1, 0.05, 0, 0, imgposn, charposn, f_hor, f_ver
```

If you want to use the default input to position the image, you can call the procedure with dummy (undefined) variables—say, `a,b,c,d,e`,f:

```
img_cbar_posns, a,b,c,d,e,f, imgposn, charposn, f_hor, f_ver
```

which assumes no colorbar; if you want the default values with a colorbar:

```
img_cbar_posns, a,b,c,d,e,f, imgposn, charposn, f_hor, f_ver, /yescolorbar
```

2.2. Create the Window

First, you need to open a window. You do this differently for your computer screen (X window, or X) and PostScript (PS):

1. For your *computer screen*, you need to generate an X-window of the proper size—with the borders and all. Suppose your image is 500×400 pixels ($X \times Y$). Then you need to generate the larger window so you have borders, which you do with the `window` command, using outputs from `img_cbar_posns`:

   ```
   loadct, 0
   window, 7, xs=f_hor*500, ys=f_ver*400
   ```

   (The `loadct,0` is a precaution, needed only if you had been using exotic color tables; it ensures that the window background is black.)

2. If you want to create a PostScript (PS) image, make sure you are using Truecolor visual class. For your PS window, you don't need to do anything except open it. We strongly recommend our `psopen` procedure, which we discuss in our tutorial **PSIDL)**.

   It is important to properly scale the aspect of the PS window, just like you scale the aspect of the X window. For the PS window you define the x- and y-sizes by `xsize` and `ysize` (see immediately below), just like you did above for the X window with `xs` and `ys`. Keeping the aspect ratio the same means

   $$\frac{ysize}{xsize} = \frac{ys}{xs}. \tag{1}$$

   So, in this case, you might choose

   ```
   psopen, 'test.ps', xsize=f_hor*5., ysize=f_ver*4., /inch,
        /color, /times, /bold, /isolatin1
   ```

   This makes the PS window size ($f\_hor * 5. \times f\_ver * 4$) inches. When opening a PostScript file for an image, the `color` keyword is necessary if you are going to use GrayScale or color; the only case when it is not needed is with pure black and white, as in a graph. Here we also have specified a particularly nice font to use on the PS image with the last three keywords; you might wish something different (and you could even revert to the somewhat clumsy Hershey fonts by eliminating all font-related keywords; see PSIDL).

3. WRITE THE DATA IMAGE ONTO THE WINDOW

This is straightforward: you use the `tv` command with appropriate keywords that define the sizes. Assume your image is a byte array named `psi_b`.

3.1. For Nondecomposed Color (a 1-d Image with a 256-element Color Table)

For nondecomposed color, you fitst load a color table (e.g., you load a gratyscale with `loadct, 0`; you load STD GAMMA II with `loadct, 5`). Then you write the image on the screen:

```
tv, psi_b, imgposn[0], imgposn[1], /norm, xsize=1./f_hor, ysize=1./f_ver
```

3.2. For Decomposed Color (a 2-d or 3-d Color Image)

Alternatively, if you are using decomposed color, which you must be doing for a 2-d or 3-d color image, then you have three images—the red, green, and blue versions of `psi_b`. You have to write all 3, which you do with the extra commands on the first line:

```
tv, [ [[psi_b_red]], [[psi_b_grn]], [[psi_b_blu]] ] , true=3, $
        imgposn[0], imgposn[1], /norm, xsize=1./f_hor, ysize=1./f_ver
```

If you will want a colorbar, you need to use the `tvlct` command to retreive the color table used in making the image and, also, to define the array `colort` for later use.

```
tvlct, r,g,b,/get
colrt= [ [r], [g], [b] ]
```

If you won't make a colorbar, you don't need these last commands.

4. IF YOU HAVE A COLORBAR, WRITE ITS IMAGE NOW!

At this point we will write only the colorbar's *image*, not its annotation.

4.1. For Nondecomposed Color (a 1-d Image with a 256-element Color Table)

For nondecomposed color, you enter

```
colorbar, pos=cbarposn, XSTYLE=4, YSTYLE=4
```

4.2. For Decomposed Color (a 2-d or 3-d Color Image)

You've already obtained the image colortable `colort` from the `tvlct` command above. To write the colorbar image:

```
loadct, 0
colorbar, pos=cbarposn, rgb=colrt, XSTYLE=4, YSTYLE=4, irange=[0,5.5], gamma=0.55
```

The second line of this call to `colorbar`, which contains the optional inputs `irange, gamma` (for which we have set arbitrary numerical values in this example), applies only for a 2-d color image, in which lightness (intensity of the image) depicts one quantity and color the other; `irange` is the data range represented by the intensity part of the image, and `gamma` is the power to which the image values are raised (as in equation 3 of **IDIDL**).

5. WRITE THE ANNOTATION ONTO THE WINDOW

5.1. Preliminaries: Define the Character Color Table

You may have written your image in vibrant color, so that each element of your 256-element color table is good for representing the *image*; or you may have written a set of three RGB images into your decomposed color PS window. Either way, you probably want a *different* set of colors for your characters and plotted lines. One way is to load a new color table and choose character colors from that.

Much easier: if you're using our startup file, then for either X or PS, just enter

```
setcolors, /system_variables
```

(or `setcolors, /sys` for short). `setcolors` provides the default colors. These include, first, the primary colors and their complements: !red, !green, !blue, !cyan, !magenta, !yellow, !white, !black; and four more nonprimary colors: !orange, !forest, !purple, !gray.

5.2. Now Annotate the Data Image!

For this you use the ordinary commands you're familiar with: `plot`, `xyouts`, etc. Suppose you just want axes and no plotting or contouring, then overwrite the axes on the image by doing

```
plot, x_b, y_b, position=imgposn, color=ps*!black+(1-ps)*!white, $
      /xsty, /ysty, /norm, /noerase, /nodata, font=ps-1...
```

Here we've assumed `x_b` and `y_b` are vectors of pixel coordinages; these are used to establish the axis min and max values[1]. Note that we specify[2] the color in such a fashion to make the axes white

[1] Alternatively, you could extablish the axis min and max values using `xra` and `yra`.

[2] *You'd better!* See §5.4.

for X and black for PS; normally this is what you want, because X has a black background and PS a white one. The `/noerase` keyword writes over the image without erasing it and the `/nodata` keyword means “don't plot the data, just put on the axes”. If you want contours instead, then enter

```
contour, psi_b, x_b, y_b, position=imgposn, color=ps*!black+(ps-1)*!white, $
      /xsty, /ysty, /norm, /noerase, font=ps-1..., ...
```

In either case, add whatever axis titles and other info you want—these procedures have *lots* of options.

At this point you might want to use IDL's `plots` procedure to plot one or more symbols somewhere on the image or `xyouts` to write a label. When you use these you should use either *data* or *normalized* coordinates, *not device* coordinates, because device coordinates are not defined for PostScript. For example:

```
xyouts, 0.3, 0.3, /normal, ORIENTATION=45, 'I AM FOREST GREEN!!', COLOR=!forest, $
      FONT=ps-1, CHARSIZE=2.0
```

5.3. Now Annotate the Colorbar Image!

We assume a 1-d colorbar and want to write its annotation in blue (God knows why!):

```
colorbar, pos=cbarposn, xtit='Velocity, Km/s', FONT=font, COLOR=!blue, /NOIMAGE
```

5.4. *SOB, SOB...* My Axes Don't Come Out Black on PS!

If you are doing esoteric things with color in PostScript, you might find that the `plot` or `contour` axes don't come out black! *Sob, sob, sob...* The solution: explicitly state the color in the call to `plot` or `contour` (by setting the keyword, for example: `color=ps*!black+(1-ps)*!white`)
.

Why does this happen? In PS, the default color for the axes is at the beginning of the color table, index 0. For most color tables, the first element is black—most images go from black to white with increasing intensity. But your color table might not do this. A good example is the `pseudo` color table, discussed in our memo **1d21d3d**: the first element is red, so the axes come out magenta unless you explicitly state black.

6. YOU'RE DONE!

You're all done! Just one thing to remember: if you're in PS, the file is empty until you close it. So after you're finished, enter

```
if ps then psclose
setcolors, /system_variables
```

7. SOME DETAILS

7.1. PS versus X: White versus Black Background

If the labels of the plot extend outside the image, then you have to be careful with the labels' colors: on the screen they are written on the screen's *black background*, but on the ps device they are written on a *white background*.

Let's reiterate that point. The default background for X is black, while that for PS is white. X uses white text on a black background; this white text on PS's white background won't show. During the process of making a pretty picture to publish, you might find it useful to give the X window a white background so that it looks like the PS background. You can do this with

```
bgfill, !white
```

As is made clear in the documentation for `bgfill`, you've just painted the window with a background color; in order to put a plot on top of this background you *must* use the `/noerase` keyword when you call `plot` (or `contour`) or you'll just overwrite the colored background (or, for PS, your plot will end up on page 2 and your colored background will be on page 1).

7.2. X Window Preliminaries: Resizing Images

Suppose you have a two-d array called `psi`, say (50×40), in which the (x, y)-coordinates are defined by (50,40)-long vectors, respectively. You need to generate a conveniently sized image for display purposes. Recall that your screen has something like 1024×768 pixels, maybe more, so the 50×40 image will occupy only a tiny area; you might (or might not) want your full image to occupy most of the screen. So perhaps you want to resize your original image to $(xsize = 500) \times (ysize = 400)$. These are integral multiples of the original size, so you can use `rebin` (as you almost always should: see below):

```
xsize = 500
```

```
ysize = 400
psi_b = bytscl( rebin( psi, xsize, ysize))
x_b= rebin( x, xsize)
y_b= rebin( y, ysize)
```

Now go to §2.2.

Comments:

1. Do as we do above: *be sure to treat the image and the coordinates identically*!

2. While you might want to increase the array size to make its X-window image bigger, you *don't* need to do this for PS. PostScript pixels are scalable, which means they will fill the designated area of the PostScript image. If you use `rebin` to increase the number of pixels in a PostScript image, the only thing you will accomplish is to increase the PostScript file size!

3. With `rebin`, the expansion or reduction factors have to be integers.

4. You can also resize the image using **congrid**, which works for non-integral factors, as follows:

   ```
   xsize = 523
   ysize = 419
   psi_b = bytscl( congrid( psi, xsize, ysize, /interp, /minus_one)
   x_b= congrid( x, xsize, /interp, /minus_one)
   y_b= congrid( y, ysize, /interp, /minus_one)
   ```

   With `congrid`, it's important to understand those last two keywords; look at `congrid`'s documentation and experiment with a 1-d array to see their effects. `/interp` makes the result smooth, while not using it retains your original pixels; `/minus_one` affects the interpolation details.

   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. In particular, *using* `congrid` *for downsizing an array always loses signal/noise. Don't use* `congrid` *unless you know what you're doing!!*

7.3. PostScript preliminaries: Use TrueColor!

As discussed in **IDIDL**, computer graphics cards can support multiple types of visual classes. If you're using Solaris or Linux, you'll want to always start IDL in TrueColor when making PS images; the reasons are beyond the scope of this document. Suffice it to say you will find oddities with your displayed images if you change a color table using either PseudoColor or DirectColor visual classes because they are *dynamic*, meaning that changing the color table changes what's already been written on the screen. The process of annotating PS usually involves changing the color table. If you use the same code for the X windows display that you do for PS, your changing the color table for the character writing will, with a dynamic color table, change the appearance of the image on the screen.