MAKING, PRINTING, AND POWERPOINTING POSTSCRIPT FILES OF GRAPHS AND IMAGES
September 10, 2011

Tim Robishaw and Carl Heiles

Contents

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

It's one thing to get your plot or image on the screen, and quite another to get a hardcopy. Getting a hardcopy always involves making a PostScript file, for which there are many options such as image size, orientation, fonts, and color. You can tackle these issues in two ways. One is to slog through the documentation and write these things yourself. The other is to take advantage of our already having done this.

There are two basic ways to generate a PostScript file. The most elegant way, which produces beautiful output, is to write the plotting commands directly to the PostScript file. The quick and dirty way is to copy your screen directly onto the PostScript file; the output looks ratty for text and graphs, which consist of lines; pixelized lines don't look very good. We emphasize the first way and give it extensive discussion in §2. We briefly treat the second way in §5.

First, however, we discuss the important issue of *fonts*. *If you don't care about fonts, skip directly to §2.*

1. A PRELIMINARY: GREEK LETTERS AND FONTS IN X and PS

1.1. Three types of font

IDL supports three types of fonts. This is done through the `font` keyword on IDL commands such as `plot` and `xyouts` that write characters to the window. These include:

1. *Hershey fonts*, invoked by setting the keyword `font=-1`; it's the default. These are vector fonts, meaning that they are drawn as a series of lines. Characters in the vector fonts are stored as equations and can be scaled and rotated in three dimensions. Being simply a series of lines, they are displayed very quickly. They were invented by Dr. A.V. Hershey of the Naval Weapons Laboratory and have nothing to do with chocolate.

   Within the set of Hershey fonts there are 16 different classes; you can see them by clicking on *fonts – about* in IDL's help index. Also, our §6.2 lists the fonts available using embedded formatting (see §1.3). The default is Simplex Roman (font number 3), which is quickest to write and looks good on the pixelized X window display. However, it is not very pretty on PS because the characters consist only of lines–there are no serifs and no thickness. They don't show up well on printed or projected images. There's also Simplex Greek (font number 4), again the quickest. You can get improved looks by invoking different font classes within the Hershey font set: for Roman and Greek, there are Duplex, Complex, Triples, and Italic versions. There are several classes for Script and Gothic, and there's Math and Miscellaneous.

   The more complex font classes take longer to write and often don't look good on the pixelized computer screen. But they do look pretty good in PostScript!

2. *Device fonts*, invoked by setting the keyword `font=0`. Device fonts, also referred to as hardware fonts, rely on character-display hardware or software built in to a specific display device and include the most important hardware font, *PostScript*. IDL includes font metric information for 35 standard PostScript fonts, and can create PostScript language files that include text in these fonts. To see these fonts, get into IDL and invoke `PS_SHOW_FONTS`: this will create a huge file called `idl.ps` in the subdirectory in which you are running IDL, which you should view in the Unix program `gv` (unless you *really* want a printed copy!). (Note: if you use `gv`, you need to specify the paper size as Letter. You can do this either in the `gv` window by changing BBox to Letter, or when you call it from Unix: `gv -media letter idl.ps`). In IDL, you can learn the details of using device fonts by using the online help: click on *fonts – about* and then *about device fonts*. Again, the fonts can be selected using embedded formatting and, also, when you open the PostScript file.

   The advantages of ps fonts are beauty, clarity, and superb quality—better than Hershey fonts and equal to or better than TrueType fonts. The disadvantage: they are supplied by the Adobe PostScript software and are *not* based on equations; therefore, they *cannot* be rotated or used in 3d graphs.

   For most purposes, these are what you should use for making ps files. Our `psopen` procedure (§1.2) makes it easy to use these elegant fonts.

3. *TrueType fonts*, invoked by setting `font=1`. We have little experience with TrueType fonts. The big advantage of TrueType fonts is that they appear identical in PS and X, so if you fiddle in X and get things looking perfect, it will work fine in PS. However, only a few of the familiar embedded formatting commands for vector fonts are available for TrueType fonts. For instance, !7 will not change the font to Complex Greek! Instead, you have to use !9. To determine which TrueType fonts IDL supports, type the IDL command
`device, GET_FONTNAMES=names, /TT_FONT, SET_FONT='*'` and print out the names array (or better yet, its transpose). The following commentary is lifted from IDL's help file.

   > TrueType fonts, also referred to as outline fonts, are drawn as character outlines, which are filled when displayed. These are generated from equations so, like Hershey fonts, can be used in 3-d plots. IDL converts the character outline information to a set of polygons using a triangulation algorithm. When text in a TrueType font is displayed, IDL is actually drawing a set of polygons calculated from the font information. Because the TrueType font outlines are converted into polygons, you may notice some chunkiness in the displayed characters, especially at small point sizes. The smoothness of the characters will vary with the quality of the TrueType font you are using, the point size, and the general smoothness of the font outlines.

1.2. Using *psopen*

To open a PostScript file you can use the native IDL command `device` and invoke all of its multifarious keywords in the proper and desirable way. You're welcome to slog through all that. Much easier is to use our non-native `psopen` command.

For most purposes, our preference is the standard Hershey fonts for X displays and device (PostScript) fonts for the ps display. Here's how we do it with `psopen`:

1. Define the variable `ps`. To produce X window output, we set `ps=0`; to produce PostScript, we set `ps=1`.

2. If `ps` equals 1, then open the ps file with `psopen`. This procedure has lots of keywords for font types; for the full set, you have to look at the procedure itself because `doc_library` gives only the essentials. The most important ones include `xsize`, `ysize`, `color`, and various font types. Our preference, which is based on opthamologist testing that shows the importance of serifs, is Bold Times Isolatin[1]...

   ```
   if ps eq 1 then psopen, filename,  /TIMES, /BOLD,  /ISOLATIN1, /color ...
   ```

3. If you have an image, write it now.

4. Before writing any characters, you need to define PostScript's color table (See our handout *Making Annotated Images* for more details):

   ```
   if ps eq 1 then setcolors, /system_variables
   ```

   The `color` keyword in `psopen` and the call to `setcolors` command are needed because, for writing characters (i.e., vector graphics), PostScript always uses PseudoColor with 256 entries and X uses whatever you have specified, which may or may not be PseudoColor if you are using Linux, Windows, or Mac.

5. When writing character strings set the keyword `font` equal to 0 for PostScript, -1 for X; the easy way is to set `font=ps-1`. (Or, if you insist on using TrueType fonts, `font=1`). For an example, see §1.3.

6. After generating the plot/image, close the ps window.

---

[1]What's this "IsoLatin"? It's the official character coding of the Latin alphabet. See *http://en.wikipedia.org/wiki/ISO_Latin-1* .

```
if ps eq 1 then begin
psclose
setcolors, /system_variables
endif
```

Note again our use of `setcolors`; we need to reset the color tables for our X window system.

1.3. Embedded Formatting and *textoidl*: A Marriage Made in Font Heaven.

There are two ways to do Greek letters and sub/superscripting. One is IDL's native embedded formatting; the other is the popular non-native procedure `textoidl`, which lets you use TEX notation. The combination is very powerful, flexible, and convenient.

The big advantage of embedded formatting is *flexibility*: you can easily change font class within a character string because you invoke a font class as part of the character string and you can do this multiple times, anywhere in the string (you invoke a class by typing its type number preceded by an exclamation point; click on *format codes – embedded in IDL strings* in IDL's help index; or see our §6). Embedded formatting also supports subscripts and superscripts and various mathematical symbols (with the Math and Special Symbols font, number 9); click on *fonts – positioning commands* in IDL's help index (or, again, see our §6).

As an example of the flexibility offered by embedded formatting, you might be creating a graph where space considerations require a small font. In such cases a conventional subscript might be too small to read. To create an ordinary smaller subscript the embedded format command is !D; to shift down *without* changing size the command is !B.

The disadvantage of embedded formatting is complexity and unfamiliarity. For this reason, most of us prefer to generate Greek letters and sub/superscripts using `textoidl`. This allows you to use the familiar TEX format to write Greek letters, subscripts, superscripts, and many of the specialized math symbols that are provided in, say, AASTEX). For a list of these, see §8.

For example, suppose we want to annotate a graph with the string $Density^2/Radius_1^3 \leq \gamma$. Using embedded formatting with Hershey fonts we have

```
xyouts, xloc, yloc, 'Density!U2!N / Radius!D1!U3!N !9l!X !4c!3', font=ps-1
```

Note to get the ≤ sign we had to convert to Font 9 (Math and Special Symbols font) by typing !9, look up how to get that symbol (by typing lowercase "l"; see §8 or the IDL documentation), and then back to the original by typing !X. In contrast, using `textoidl` we have

```
xyouts, xloc, yloc, textoidl('Density^2 / Radius_1^3 \leq \gamma', font=ps-1), font=ps-1
```

They produce identical output. You could, if you really wanted, put the embedded positioning commands within the `textoidl` string instead of using the TEX-style subscripting and superscripting.

IMPORTANT: You must specify the font keyword (`font=ps-1`) in *both* `textoidl` *and* `xyouts`!!!!! For ps fonts, you must have `font=0`; and for Hershey fonts, you must have `font=-1`. And you must do this in *any* plot command that writes characters, such as `plot` and `contour`. The *easy* way to do this is, before starting your plotting, *define the system font variable*; then you don't have to define it in each plotting command. To do this, enter

```
!p.font = ps - 1
```

When you are all done with plotting, you should prabably set it back to the hardware font value,

```
!p.font = -1
```

1.4. A Cautionary Note about Embedded Font Selection

Embedded font selection can be very useful, but there is a problem: the meaning of a font number depends on whether you are using Hershey, hardware, or TrueType fonts! See the table in §6.2. This is a problem because your character output will depend on the value of the `font` keyword.

This is the beauty of `textoidl`: it does Greek and Roman seamlessly without changing the `font` keyword. So for most purposes, `textoidl` is much better than embedded font selection!

1.5. A Positive Note about Font Selection

Once you specify a font, the specification remains unless you change it. In particular, this applies to embedded fonts—meaning that if you change the font when writing a character string with `xyouts` for example, it will remain changed in future calls to `xyouts`. If you don't want this, you must revert back to the original font, which you could do with the !X embedded font command.

This is also true with PostScript font selection: once you call `psopen` with a set of fonts, they will remain in future calls unless you specify otherwise.

2. ELEGANCE AND BEAUTY: PLOTTING DIRECTLY TO THE PS DEVICE

Here you specify the ps device as the output window and use IDL's plotting commands (like `plot`, `xyouts`, and `tv`) to write directly to it. This takes full advantage of PostScript's unique ability to deal with the two basic types of image, *vector graphics* and *pixel images*. If you do it right, you also gain the flexibility and built-in excellence of PostScript fonts.

Normally, you first generate your plot or image on the X terminal window, going through a process of iteration until you arrive at what you want. This involves invoking a set of IDL commands. Having done this, you specify PostScript as the output device and invoke the same set of IDL commands. You look at the resulting plot or graph using the UNIX `xv` or `gv` program (we recommend `xv`). You'll find that it doesn't look exactly the same as it did on X, so you'll need to do some more fiddling (unless you're using TrueType fonts).

So making and looking at the ps file is normally a four-step process:

1. Define the PostScript device as the output device. We recommend using `psopen`; see §1.2.
2. Generate the plot or image, with annotation.
3. Close the ps device and redefine the output device as the X-window. Again, see §1.2.
4. Look at the output with `xv test.ps` or `gv test.ps` and, if necessary, make iterative adjustments by returning to step 2.

3. PRINTING COLOR IMAGES

Color laser printers are the rage. However, their ability to display colors is *very limited.* They are fine for graphs with different colored lines and other applications where there are no subtle color issues. Otherwise they are terrible.

Inkjet printers are much, much better at reproducing color. They can do very well with complicated color images. That's why all printers meant for photographic reproductions at home are inkjets. Our department has both a laser and an inkjet color printer. To use the laser printer, type `lp -d color filename.ps` . To use the inkjet:

```
lp -d photo filename.ps
```

When this printer doesn't work (note the "when", not "if") go see Kelley or Bill. Of course, if you are using transparencies, you are a dinosaur. Try OpenOffice or PowerPoint! (Carl is a dinosaur but nevertheless has been propelled into the modern age, kicking and screaming, by the restrictive requirement, at scientific meetings, of providing PowerPoint talks on a memory stick.)

4. POWERPOINTING POSTSCRIPT IMAGES

You can't import a PostScript file into PowerPoint. So what's the poor Linux user to do? Use the Linux `convert` program to create a `.png` file and create plenty of pixels so you don't lose resolution:

```
convert -verbose -antialias -density 600 -geometry 25% -quality 95
file.ps file.png
```

When you import the result into PowerPoint, you'll have to make the image smaller. But it's worth the extra effort—the result will look superb! The reason: `png` files are *lossless.* `png` stands for "Portable Network Graphics"; see `http://www.libpng.org/pub/png/`

One more thing. The native resolution of most projectors is 1024×768. You should set your laptop's output display size to this. Otherwise you risk losing the edges of your image or having little waves crawling up the display.

5. QUICK AND DIRTY: WRITING SCREEN PIXELS TO POSTSCRIPT

Above we described how to make elegant, beautiful PostScript images. It requires rerunning the plot/imaging commands. Here we describe the quick and dirty way, which does *not* require rerunning: you first generate the X window version; then you read the image pixels directly from the screen and turn them into a PostScript file. Using our procedures, this is quick and totally painless. However, the output looks ratty for text and graphs, which consist of lines; pixelized lines don't look very good. But you may be willing to put up with this sometimes—if you're in a hurry, or making a hardcopy for your lab notebook, for example. If you want to use this quick and dirty technique but want better-looking results, use a larger window; the pixelization on the hardcopy will be less noticeable.

This consists of two subsections, one for plots and one for images. For plots, we assume grayscale with either two levels (1 bit—black and white) or 256 levels (8 bits—grayscale; some plots have shading). If your color table has fewer than 256 levels, we interpolate it to 256; this is great for grayscale, but if you are using a non-gray color table it will probably give you weird results.[2]

Both procedures retain the aspect ratio on the window, even if you try to change it with the keywords. If you want a different aspect ratio, then generate a new window with the desired aspect ratio (using IDL's **window, xsize=256, ysize=512**, for example) or rewrite the procedure for yourself.

---

[2]If this statement confuses you, see **BIDIDL**.

5.1. Copying plots with *hardplot*

We assume you've already generated your plot. Specify the window from which to read the image using IDL's **wset** command. Then type

hardplot

It will ask for the name of the output filename. **hardplot** is a home-grown procedure that has keywords that allow you to do various things. The default values are set for reproducing ordinary black/white plots, *inverting the white-on-black that you see on your screen to the black-on-white that you should use for a printed output.* In particular, it produces *only* black on white in the ps file; so if the original had, say, red and yellow and white graphs on the X black background, the three lines would all appear black against the white ps background. See the documentation.

Suppose you've called the resulting PostScript file *plot.ps*. To view this PostScript file before printing—always a good idea—use the UNIX command

xv plot.ps

or, from within IDL, you can invoke this (or any other) UNIX command by putting a dollar sign in front of it:

$xv plot.ps

The *xv* utility has lots of options that you can access by moving the mouse cursor onto the image and clicking the right-hand button. Try it!

5.2. Copying images with *hardimage*

`hardimage` reproduces the X window display faithfully to the PostScript file. It's almost identical to `hardplot` running with *nbits=8* and the *noreverse* option, but it also copies the colors if the image is not grayscale.

6. APPENDIX 1: EMBEDDED FORMATTING COMMANDS

Here we provide the embedded formatting commands used by IDL. To use them, simply write the command preceded by an exclamation point; the formatting remains until you change it[3].

---

[3]So how do you write an exclamation point? You type *two* exclamation points!!

6.1. Positioning Commands

We begin with the positioning commands:

```
Command Action
!A  Shift above the division line.

!B  Shift below the division line.

!C  "Carriage return," begins a new line of text.  Shift back to the
starting position and down one line.  This command also performs an
implicit "!N" command, returning to the normal level and character size
at the beginning of the new line.

!D  Shift down to the first level subscript and decrease the character
size by a factor of 0.62.

!E  Shift up to the exponent level and decrease the character size by a
factor of 0.44.

!I  Shift down to the index level and decrease the character size by a
factor of 0.44.

!L  Shift down to the second level subscript.  Decrease the character
size by a factor of 0.62.

!N  Shift back to the normal level and original character size.

!R  Restore position.  The current position is set from the top of the
saved positions stack.

!S  Save position.  The current position is saved on the top of the saved
positions stack.

!U  Shift to upper subscript level.  Decrease the character size by a
factor of 0.62.

!X  Return to the entry font.

!Z(u0,u1,...,un) Display one or more character glyphs according to their
unicode value.  Each ui within the parentheses will be interpreted as a
```

16-bit hexadecimal unicode value. If more than one unicode value is to be included, the values should be separated by commas.

!! Display the ! symbol.

6.2. Font Commands

We finish with the font commands. The meaning of the font commands depends on whether you are using Hershey, hardware, or TrueType fonts, as outlined in the table below:

Command	Vector Font	TrueType Font	PostScript Font
!3	Simplex Roman (default)	Helvetica	Helvetica
!4	Simplex Greek	Helvetica Bold	Helvetica Bold
!5	Duplex Roman	Helvetica Italic	Helvetica Narrow
!6	Complex Roman	Helvetica Bold Italic	Helvetica Narrow Bold Oblique
!7	Complex Greek	Times	Times Roman
!8	Complex Italic	Times Italic	Times Bold Italic
!9	Math/special characters	Symbol	Symbol
!M	Math/special characters (change effective for one character only)	Symbol	Symbol
!10	Special characters	Symbol *	Zapf Dingbats
!11(!G)	Gothic English	Courier	Courier
!12(!W)	Simplex Script	Courier Italic	Courier Oblique
!13	Complex Script	Courier Bold	Palatino
!14	Gothic Italian	Courier Bold Italic	Palatino Italic

!15	Gothic German	Times Bold	Palatino Bold
!16	Cyrillic	Times Bold Italic	Palatino Bold Italic
!17	Triplex Roman	Helvetica *	Avant Garde Book
!18	Triplex Italic	Helvetica *	New Century Schoolbook
!19		Helvetica *	New Century Schoolbook Bold
!20	Miscellaneous	Helvetica *	Undefined User Font
!X	Revert to the entry font	Revert to the entry font	Revert to the entry font

* The font assigned to this index may be replaced in a future release of IDL.

6.3. Defining your own font in PostScript

Suppose you are using PostScript fonts and want to use the Bookman Demi Italic with ISO-Latin1 font encoding for one of your math symbols. This is not one of the current embedded formatting options. Nevertheless, you can do it if you are using PostScript fonts! IDL lets you replace one of the font indices with whatever font you define.

Here's how you do this: Let's replace the "Zapf Dingbats" PS font in the !10 font index.

```
IDL> psopen, 'foo.ps', /HELVETICA, /BOLD, /OBLIQUE, /ISOLATIN1
IDL> DEVICE, /BKMAN, /DEMI, /ITALIC, /ISOLATIN1, FONT_INDEX=10
IDL> plot, findgen(3), FONT=0, $
IDL>   xtit='Galactic Radius !10'+string(174B)+'!X [kpc]', $
IDL>   ytit='Density !10'+string(181B)+'!X [cm!E-3!N]'
IDL> psclose
```

Note the use of !10, which converts to your defined font; and !X, which converts back to the original one[4]. You *cannot* change font indices like this for TrueType or vector fonts! Only PostScript fonts!

[4]Note, also, we didn't use the *color* keyword and call to *setcolors* because this is a pure black/white (no gray or

7. APPENDIX 2: CHARACTER SIZE AND LINE SPACING

When annotating graphs or images with `xyouts`, the bottom of the characters locate at the y value you specify. Sometimes you want the *middle* of the characters to so align. To do this, you need to align the character lower by half its height. How do you know the character height?

Look in the online help under `SET_CHARACTER_SIZE`; it tells you about the character sizes, which are stored in `!D.X_CH_SIZE` and `!D.Y_CH_SIZE`. The units are device pixels. The default pixel size in PostScript is 10^{-3} cm, so this is kind of awkward. . . .

It's probably your choice to place the text in data coordinates instead of device or normalized coordinates. The text height in data coordinates is

```
y_ch_size_data = !d.y_ch_size / !y.s[1] / !d.y_vsize
```

To make the center of the symbols align where you want in the vertical direction, subtract half of this from the desired y position.

8. APPENDIX 3: MATH SYMBOLS SUPPORTED BY *textoidl*

Here is a table of the specialized math symbols supported by our version of `textoidl`. The first column gives the `textoidl` sequence, the second the character in font number 9, and the third the specificaion in PostScript if you are not using `textoidl`.

```
TeX SEQUENCE     VECTOR     POSTSCRIPT

\aleph             @      string(byte(192))
\ast               *             *
\cap               3      string(byte(199))
\cdot              .      string(byte(215))
\cup               1      string(byte(200))
\exists            E             $
\infty             $      string(byte(165))
\in                e      string(byte(206))
\equiv             :      string(byte(186))
\pm                +      string(byte(177))
\div               /      string(byte(184))
\subset            0      string(byte(204))
```

---

colors) image.

`\superset`	`2`	`string(byte(201))`
`\leftarrow`	`4`	`string(byte(172))`
`\downarrow`	`5`	`string(byte(175))`
`\rightarrow`	`6`	`string(byte(174))`
`\uparrow`	`7`	`string(byte(173))`
`\neq`	`=`	`string(byte(185))`
`\propto`	`?`	`string(byte(181))`
`\sim`	`A`	`string(byte(126))`
`\partial`	`D`	`string(byte(182))`
`\nabla`	`G`	`string(byte(209))`
`\angle`	`a`	`string(byte(208))`
`\times`	`X`	`string(byte(180))`
`\geq`	`b`	`string(byte(179))`
`\leq`	`l`	`string(byte(163))`
`\'`	`'`	`string(byte(162))`
`\prime`	`'`	`string(byte(162))`
`\circ`	`%`	`string(byte(176))`

For more complex symbols you will need to use embedded fonts. Alternatively, you can define your own in `textoidl`; see its documentation.