

# $\pi$ -GPLOT (pygplot) version 0.91

Chris Burns

July 12, 2007

## 1 Background

Pygplot is a python module which provides the user with a friendly, object-oriented interface to the PGPLOT plotting library. It arose out of my first attempt at writing a complete object-oriented package. PGPLOT is a FORTRAN (and C) library of functions which are used to produce scientific, journal-quality plots from data<sup>1</sup>. It requires the user call several functions, each performing a separate task in setting up a plot (e.g., PGBEGIN opens the graphics device, PGENV sets the ranges and draws the axes, PGLINE draws a line, etc). Because it is written in, and primarily intended for FORTRAN, the library does not immediately lend itself to interactive plotting. Also, due to the limitations of FORTRAN (or C), there is no simple way to have default function arguments and so no easy way to have a one-function-does-it-all interface.

Luckily, with the advent of scripting languages like python, perl and tcl, one can wrap the FORTRAN functions in a more user-friendly guise. Nick Patavalis of the University of Athens ([npat@ariadne.di.uoa.gr](mailto:npat@ariadne.di.uoa.gr)) did just that and provided a python module which would allow python to call PGPLOT functions directly. He even provided simplified versions of the PGPLOT functions, so that the user need not provide the sizes of input arrays, for example. However, this was still very much like calling PGPLOT functions from C or FORTRAN, only it was done from within python. Scott Ransom ([ransom@cfa.harvard.edu](mailto:ransom@cfa.harvard.edu)) then took these raw functions and wrapped them in a more user-friendly one-function-does-it-all interface. I used these functions for some time, but finally felt that one should really have an object-oriented interface to the library, but which would still lend itself to interactive plotting.  $\pi$ -gplot is the result and evolved from hacking Scott Ransom's package to pieces and putting them in a class structure. One common aspect to all these packages is that anywhere PGPLOT expects an array, you need to use a Numeric array. Numeric is the standard way to work efficiently with arrays in python. There is also a new package called numarray which will soon take over, but it will work just as well with  $\pi$ -gplot.

For the rest of this guide, it is assumed the reader has a working knowledge of python. Understanding classes is not necessary, but will probably help in some of the discussion. A casual browsing of the PGPLOT manual is also recommended.

## 2 A Simple Example

To give the user an idea of what  $\pi$ -gplot offers, let's look at a simple example of a plotting session and how it would be accomplished in FORTRAN and then with  $\pi$ -gplot. Suppose

---

<sup>1</sup> PGPLOT is freely available for non-commercial use and is copyrighted by the California Institute of Technology. See their webpage at <http://www.astro.caltech.edu/~tjp/pgplot/>

we have an array X and Y and wish to plot one versus the other. In FORTRAN, you would need to know the size of the arrays and also their minimum and maximum values so that you know what range to use when plotting. Let's say that N is the size of X and Y and that XMIN, XMAX, YMIN and YMAX are the limits of the graph. We want a title, say "X versus Y" and x and y axis labels: "X" and "Y". For now, we can just connect the data with lines. The FORTRAN code would look something like:

```
PGBEG(0, '/XWIN', 1, 1)
CALL PGENV(XMIN, XMAX, YMIN, YMAX, 0, 1)
CALL PGLAB('X', 'Y', 'X versus Y')
CALL PGLINE(N, X, Y)
CALL PGEND
```

Notice that all the information about the plot must be provided. There is no default behaviour, no way to just say "plot X versus Y" and use an x- and y-range that makes everything fit. For more complicated behaviour, there will be more function calls. Also, if you want to make any changes, you need to re-compile your program and re-run it. Now let's look how this would happen in  $\pi$ -gplot:

```
import pygplot
plot = pygplot.Plot(title="X versus Y", xlabel="X", ylabel="Y")
plot.line(x,y)
plot.plot()
```

While it doesn't look like we've gained in the number of calls to functions, we have gained a lot of versatility. Notice that nowhere do I need to know the size of the array x and y, nor do I need to specify a range for x and y:  $\pi$ -gplot handles that automatically. Of course, I could always force the issue by telling the `Plot()` constructor what the range is. You could also go back and change the `xrange` interactively:

```
plot.xrange = [0.0, 10.0]
plot.replot()
```

then when you were happy with the preview window, close the display, change the device to a postscript file, say, and plot it out:

```
plot.close()
plot.device="output.ps/PS"
plot.plot()
plot.close()
```

and you now have the file called `output.ps` that you can print.

### 3 The Plot Constructor and its Options

The basic object in  $\pi$ -gplot is the `Plot` class. Think of it as a container into which you deposit x-y lines, points, symbols, contours, etc. When it's all setup, you call its `plot()` function. Then use the `close()` function to clean up and make sure the output buffer is flushed. When you call the constructor to create a `Plot` instance, you can give many options which will control the overall look or the plot. Table 1 summarizes the options which can be set and their default values.

option	Summary	Default
aspect	The aspect ratio (height/width) of the plot. 1.0 produces a square plot	Device specific
color	Color with which objects are drawn (lines, symbols, etc).	Foreground color
device	PGPLOT device to which the plot is sent	'/XWIN'
font	Font with which to print labels, title, etc.	1
fsize	Font size with which to print the labels.	1.0
id	Whether or not to print the user ID under the plot (1=yes, 0=no)	0
linestyle	Line style of the data lines and arrows.	1
linewidth	Width of data lines.	1
noleftlabel, norightlabel, notoplabel, nobottomlabel	Sometimes the first/last tick labels on the x/y axes overlap. Set any or all of these to 1 to suppress the first tick labels.	PGPLOT default behaviour
symbol	Default symbol to use for plotting points.	1
ticks	Draw ticks 'in', 'out' or 'both'.	'in'
title	The title of the plot	None
width	Width of all other lines in the plot (axes, arrows, etc)	1
xaxis, yaxis	Whether or not to draw the x or y zero-axes.	0
xformat, yformat	The format to plot the x and y tick labels. If this is "dms", the tick values are interpreted as arc-seconds and are converted into degrees, minutes and seconds. If "hms", tick values are interpreted as seconds and labels are converted to hours, minutes and seconds.	Normal decimal numbers
xgrid, ygrid	Draw a grid at major tick marks in the x and y directions. Set this to the color of the grid.	No grid is drawn
x2format, y2format	Same as xformat and yformat, but for the secondary axes.	Normal decimal numbers
xrange, yrange	The x and y range to plot (both should be 2 element lists).	auto range
x2range, y2range	The x and yrange for the secondary axes.	auto range
xlabel, ylabel	The x and y labels.	None
x2label, y2label	The secondary x and y labels.	None
xlog, ylog	Whether or not to use logarithmic axes for x and/or y axes.	0
x2log, y2log	Whether or not to use logarithmic secondary axes.	0

Table 1: Options to the Plot() constructor.

## 4 Common Attributes

Many of the PGPLOT objects use attributes that are common. Here, I cover several ones which will come back again and again.

### 4.1 Color

Each PGPLOT device has a number of colors that are available. Typically, there are 16 colors available for plotting lines and labels, but there could be more or fewer (e.g., a B/W printer has only two colors). The color 0 is the background color and 1 is the foreground color and is usually the default. In  $\pi$ -gplot, you can also specify colors by their string representations: 'red', 'green', 'blue', etc. Table 2 shows the first 16 colors defined in most color tables and their string equivalents. Note, however, that the string 'black' is set to the background value 0 and 'white' is set to the foreground value 1. This is the way the X-windows viewer is setup. However, on other devices (PS, for example), it will be the opposite. For this reason, there is also an 'fg' and 'bg' string which may be less confusing.

Number	String	Number	String
0	'black' or 'bg'	8	'orange'
1	'white' or 'fg'	9	'green2'
2	'red'	10	'green3'
3	'green'	11	'blue2'
4	'blue'	12	'purple'
5	'cyan'	13	'pink'
6	'magenta'	14	'darkgray'
7	'yellow'	15	'lightgray'

Table 2: Standard colors in the color table and string equivalents.

### 4.2 linestyle

Most line objects take this option. There are 5 linestyles to choose from:

1. full line,
2. long dashes,
3. dash-dot-dash-dot,
4. dotted,
5. dash-dot-dot-dot.

### 4.3 linewidth

This option is an integer and specifies the width of a line in multiples of 0.005 inches (0.13 mm). The exact result depends on the device.

## 4.4 fsize and size

Fsize controls the font size of any labels and size controls the size of symbols. They are specified with a floating point number in units of the default height (which is 1/40th of the width or height of the display, whichever is smaller).

## 4.5 font

This controls the font used to plot labels. It's value can be an integer between 1 and 4, corresponding to PGPLOT's 4 fonts:

1. normal font
2. Roman font
3. Italic font
4. Script font

# 5 X-Y Plots

Once the Plot instance has been created, you add content by calling its member functions. The simplest of these are the ones which produce 2-D plots (x vs. y). They require two arrays: the x and y values to plot. The member function `line()` connects the data with a line, `symbol()` plots the data with symbols and `error()` lets you draw error bars.

## 5.1 Line Plots

The `line()` function takes an array of x values and y values and draws lines between them. There are a number of options that can be set to change the appearance of the lines: `linestyle`, `linewidth` and `color`. If you don't specify these options, they are inherited from the `Plot()` class.

## 5.2 Point Plots

The `point()` function plots symbols at the coordinates defined by the required x and y arrays. The `point()` function has the following options: `color`, `size` and `symbol`. The `symbol` option is an integer corresponding to the desired symbol (see PGPLOT documentation for a table).

## 5.3 Color Point Plots

Suppose you want to plot X-Y points, but you want their color to reflect a third quantity. This can be done using the `color_point()` function. It requires 3 arrays: x, y, and z (which controls the color). The translation between z-value and color is done in the same manner as the `image()` routine, using a color palette, so one of the arguments is for the color palette and transfer function (see section 7 for details about palettes and transfer functions).

## 5.4 Error Bars

The `error()` function is used to plot errorbars. The required arguments are two arrays, corresponding to the centers of the error bars. One can specify up to four other arrays with the same length as `x` and `y`: `dx1`, `dx2`, `dy1` and `dy2`. If only `dx1` is specified, the errorbars are drawn with a width of `dx1` in the positive and negative directions (total width of  $2 \times dx1$ ). If `dx1` and `dx2` are both specified, horizontal errorbars are drawn with upper limits given by `dx1` and lower limits given by `dx2`. Similar rules apply to `dy1` and `dy2`. The options `linestyle`, `width` and `color` can also be specified with obvious effects.

## 5.5 Histograms

The `histogram()` function can be used to bin and plot a histogram of data. The only required argument is a vector of data. The routine will then figure out the bins (by default, 10 bins equally spaced between the largest and smallest value of the input data) and draw the rectangles representing the histogram. There are three arguments which control the bins: `nbin`, `low` and `high`. The routine will bin the data into `nbin` bins between `low` and `high`. If an element of the data lies on a boundary, it is placed in the upper bin. The `histogram()` function also takes the usual arguments `color`, `linestyle` and `linewidth`, which control the outline of histogram. To control the filling of the bars, use `fill`, `fillstyle`, and `fillcolor`. The argument `fill` is simply a boolean: whether or not to fill the bars. The `fillstyle` argument controls how the filling is done (1- solid, 2- outline, 3- hatched, 4- cross-hatched). `fillcolor` will determine the color used to fill the bars.

## 5.6 Example

Here is a complete example of a python session which creates some data and plots it out using symbols, lines and error bars.

```
>>> import pyplot
>>> import Numeric
>>> x1 = Numeric.arange(100)*2*Numeric.pi/100.0
>>> x2 = Numeric.arange(10)*2*Numeric.pi/10.0
>>> y1 = Numeric.sin(x1)
>>> y2 = Numeric.sin(x2)
>>> pl = pyplot.Plot(title="An Example Plot", xlabel="x", ylabel="y",
...                 xrange=[0,6], yrange=[-2,2])
>>> pl.line(x1,y1,color='red', width=2, linestyle=3)
>>> pl.point(x2,y2, color='yellow', size=2.0, symbol=5)
>>> pl.error(x2,y2, dx1=y2*0.1, dy1=x2*0.1, width=2)
>>> pl.plot()
>>> pl.close()
```

This session produces the graph shown in Figure 1, which by default will be plotted on an `xwindow`.

## 5.7 Labels

There are several types of labels in PGPLOT: title, axis labels and labels which can be placed arbitrarily within the plotting area. We've already seen how titles and axis labels are

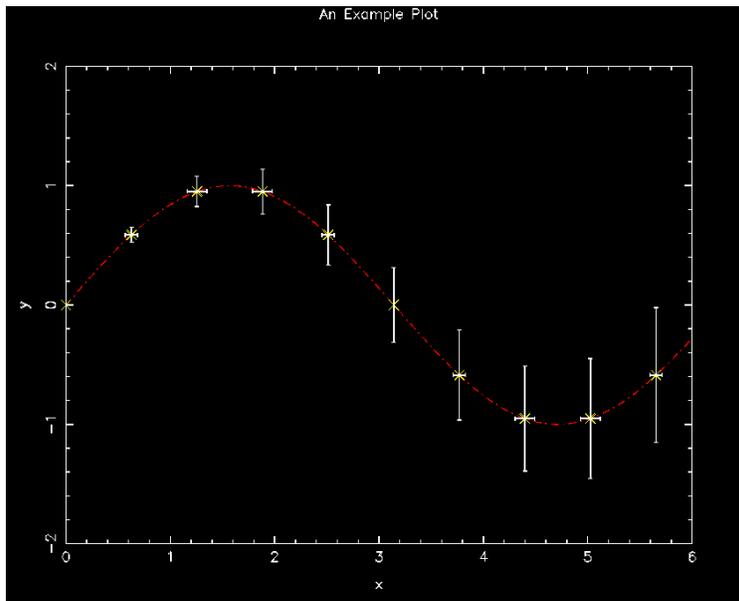


Figure 1: Graph produced by the example of section 5.6

produced. To add other labels in a plot, use the `label()` member function. This function has 3 required arguments: the (x,y) position of the label and the string to plot. Three other arguments have already been discussed: font, color and size. There are two other arguments which can be provided: `just` and `angle`. This argument just controls the justification of the label relative to the (x,y) coordinates and ranges from 0.0 (left justified) to 1.0 (right justified). The argument `angle` allows the label to be printed at an angle relative to the x-axis (in degrees).

**New in 0.91:** you can specify a `width` for the label (in world coordinates). If the label is wider than this width, it will be split over multiple lines. In this case the option `vjust` will control how the multiple lines are vertically justified. If `vjust` is 0, the bottom of the entire label is aligned at (x,y). If `vjust` is 1, the top is aligned with (x,y). `vjust` can also be any fraction in between.

There is also a `bbox` argument that can be set to make a bounding box around the label. If `bbox` is a single color, a simple rectangle is drawn around the label. If `bbox` is a list of 2 colors, the first is used to draw a bounding rectangle and the second is used to fill in that rectangle before printing the label.

All labels in PGPLOT can contain special escape characters which allows the user to mix roman and Greek letters, special characters, and use subscripting and superscripting. This is all well documented in the PGPLOT manual and will not be reproduced here<sup>2</sup>.

## 5.8 Other Axes

You can add other axes at arbitrary orientations and scalings by using the `axis()` function. This function requires 4 floats: `x1`, `y1`, `x2`, and `y2`, which correspond to the coordinates of the two ends of the axis. There are the usual optional arguments: `fsize`, `font`, `linestyle`, `linewidth`, and `color`. There are also several other optional arguments that control how the axis is drawn:

---

<sup>2</sup> The manual is available in both online and printed format at <http://www.astro.caltech.edu/~tjp/pgplot/contents.html> Chapter 4 deals with text labels

- **dec**: force decimal tick labels
- **disp**: displacement of tick labels away from the axis in units of character height (default is 0.3).
- **dmajl**: length of the major tick to the left of the axis in units of character height.
- **dmajr**: length of the major tick to the right of the axis.
- **exp**: force exponential tick labels
- **fmin**: length of the minor tick in units of the major tick.
- **nsub**: number of minor ticks per major tick interval.
- **orient**: angle at which to print labels relative to the axis.
- **step**: major tick spacing along the axis.

## 5.9 Legend

New in version 0.9 is the ability to make a legend automatically. This is accomplished in a two-step process. First, when you plot points or lines using the `line()` and `point()` functions, specify a label using the `label` argument. Any data with a `label` specified will be included in the legend.

Next, you simply call the `legend()` member function before calling the `plot` function. There are, of course, several options to specify how the legend is plotted. The usual arguments `font`, `fsize`, and `color` have obvious effects.

The legend is drawn relative to one of the four corners of the plot. By specifying `position`, you can change this. The default is to put the legend in the upper-right (`position = 'ur'`), but you can also choose lower-right (`lr`), lower-left (`ll`), or upper-left (`ul`). You can specify how far the legend should be offset from the axes by specifying `dx` and `dy` (in world coordinates). You can also impose a maximum `height` and `width` for the legend. If the legend is larger than the maximum width, the lines are split. If the legend is larger than the specified height, the font is scaled to make it fit. Lastly, you can specify a bounding box be drawn around the legend. Use the optional argument `bbox` in the same way as the `label` command (see section 5.7).

## 6 Contour Plots

Contour plots are constructed in much the same way as x-y plots, however more information may be needed in this case. First, the `contour()` function requires a 2 dimensional array (matrix). In order to contour this data, we need to know at what levels to contour and also how to convert pixel coordinates into the world coordinates of the graph. If no other information is given to `contour()`, the contour levels will be computed automatically: ten contours, equally spaced between the lowest and highest pixel values. Likewise, the default world coordinates will be equivalent to the pixel coordinates. More often, you will want to control the contour levels and world coordinate transformations, which is described in the next two sections.

## 6.1 Contour Levels

There are several ways to specify the contour levels. The most direct method is to simply specify the levels as a list or array, using the `contours` argument. The other ways are more automatic. First, one can simply specify the `low`, `high` and `ncontours` arguments. In this case `ncontours` levels, equally spaced between `high` and `low` will be used. Lastly, one can also specify a user-defined function by way of the `cfunc` argument, which will generate the contour levels. The function should take 3 arguments: the value of `low`, `high` and `ncontours`. It should then return a tuple of contour levels. This function will then be called automatically by the `plot()` function.

## 6.2 World Coordinates

The array of data provides floating point values at each pixel coordinate (`xp`, `yp`). However, in order to draw the contours, we need to be able to convert from pixel coordinates to world coordinates (`x`, `y`). There are two ways to do this.

First, one can specify the arguments `xrange` and `yrange`. Both must be 2-tuples which correspond to the range of the `xp` and `yp` axes of the array. This need not be the same as the `xrange` and `yrange` of the graph itself. This allows for an arbitrary scaling and translation of the coordinates, but no rotation or shear.

Second, one can specify a 6-tuple of transformation coefficients through the argument `tr`. Given pixel coordinates (`xp`, `yp`), the world coordinates are given by

$$\begin{aligned}x &= \text{tr}[0] + \text{tr}[1]*\text{xp} + \text{tr}[2]*\text{yp} \\y &= \text{tr}[3] + \text{tr}[4]*\text{xp} + \text{tr}[5]*\text{yp}\end{aligned}$$

A non-zero value for `tr[2]` and/or `tr[4]` will produce shear and/or rotation of the coordinate system.

## 6.3 Contour Labels

One can also have the contours labeled. How this is done is controlled by the arguments `clab`, `clabc`, `clabi` and `clabm`. If `clab` is non-zero, the contours will be labeled. The argument `clabc` specifies the color of the label. Labels will be spaced `clabi` grid cells away from each other along the contour (see the PGPLOT manual for more info on this). Contours which cross less than `clabm` grid cells will not be labeled. The default values of these latter two arguments are 40 and 10, respectively. The arguments `font`, `fsize` and `color` control the appearance of the labels.

## 6.4 Example

Here is a python session that creates a two-dimensional array and draws contours of that data.

```
>>> import pygplot
>>> import Numeric
>>> from math import *
>>> def distance(x,y):
...     return(Numeric.sqrt(x**2+y**2))
...
...

```

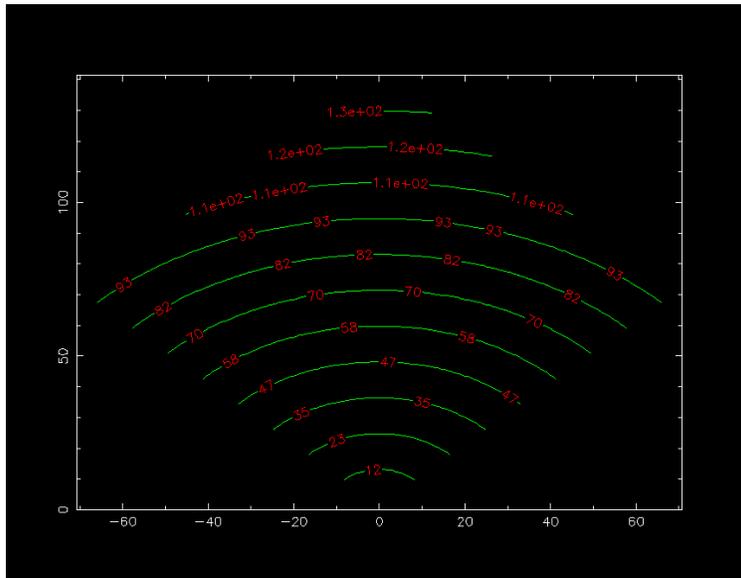


Figure 2: Graph produced by the example in section 6.4

```
>>> a = Numeric.fromfunction(distance, (100,100))
>>> tr = [0.0, cos(pi/4), -sin(pi/4), 0.0, sin(pi/4), cos(pi/4)]
>>> pl = pygplot.Plot()
>>> pl.contour(a, tr=tr, ncontours=12, color='green', clab=1, clabc='red')
>>> pl.plot()
>>> pl.close()
```

Figure 2 shows the results of this example.

## 7 Images

PGPLOT allows one to plot a two-dimensional array as a grey-scale or color image on the graphing area. The syntax for this is similar to the contouring function, but instead of contour levels, you need to specify a color map, or palette, to tell PGPLOT how pixel values are mapped to a specific color. Use the `Plot()` instance's `image()` function to work with images. It's only required argument is the array to be displayed.

### 7.1 Color Palette

Pygplot comes with its own set of palettes to choose from. They are referred to by a string representation. Use the `pygplot.list_palettes` function to display a list of the available built-in palettes. The default is "real". The palette is specified by using the `palette` argument to `image()`. By default, the lowest pixel value will be mapped to the lowest color in the palette, the highest pixel value is mapped to the highest color in the palette and every other pixel value is mapped to its appropriate color in the table by linear interpolation. Similar to the `contour()` function, `image()` accepts a `high` and `low` argument. Any pixel value higher than `high` is mapped to the highest color and any pixel with value lower than `low` is mapped to the lowest color. This way, you can "cut" out high-end and low-end pixel values.

You can also specify a non-linear map between pixel value and color by specifying the `transfer` argument. There are several options which are also listed with `pygplot.list_palettes()`. You can also run the test program `test_colormaps.py` to get a table of all the different palettes and transfer functions.

There is also an `autocut` attribute that you can use to automatically choose the `low` and `high` cuts. Set `autocut` to the percentage of pixels you wish to be included between `low` and `high`. This is especially useful in astronomical images, where there might be bright point sources which would normally dominate the color map, keeping you from seeing the lower brightness structure in the image.

## 7.2 World Coordinates

This is specified exactly the same way as with the contours, either by specifying `tr` directly, or by specifying the `xrange` and `yrange` arguments. See section 6.2 for more details.

## 7.3 Example

In this example, we extend the previous example in section 6.4 by adding a color image on the background.

```
>>> import pygplot
>>> import Numeric
>>> from math import *
>>> def distance(x,y):
...     return(Numeric.sqrt(x**2+y**2))
...
>>> a = Numeric.fromfunction(distance, (100,100))
>>> tr = [0.0, cos(pi/4), -sin(pi/4), 0.0, sin(pi/4), cos(pi/4)]
>>> pl = pygplot.Plot()
>>> pl.contour(a, tr=tr, ncontours=12, color='green', clab=1, clabc='red')
>>> pl.image(a, tr=tr, palette='astro')
>>> pl.plot()
>>> pl.close()
```

Figure 4 shows the results of this example.

## 8 Panels

The native PGPLOT allows for multiple plots on a single device “page”. Essentially, it subdivides each page into sub-pages and plots full graphs (complete with title and axis labels) in each sub-page. However, it is often desirable to have multiple plots on the same page, but with adjacent graphs sharing axes. Figure 4 shows the difference between the two cases. The left-hand plot was produced using the `MPlot()` class and the right-hand one was produced using the `Panel()` command. Both classes work the same way: you produce a `Panel` or `MPlot` instance and use its `add()` member function to add `Plot` instances. When you have filled all the sub-plots, you run the `Panel` or `MPlot` instance’s `plot()` and `close()` functions. Figure 4 shows the results of this example.

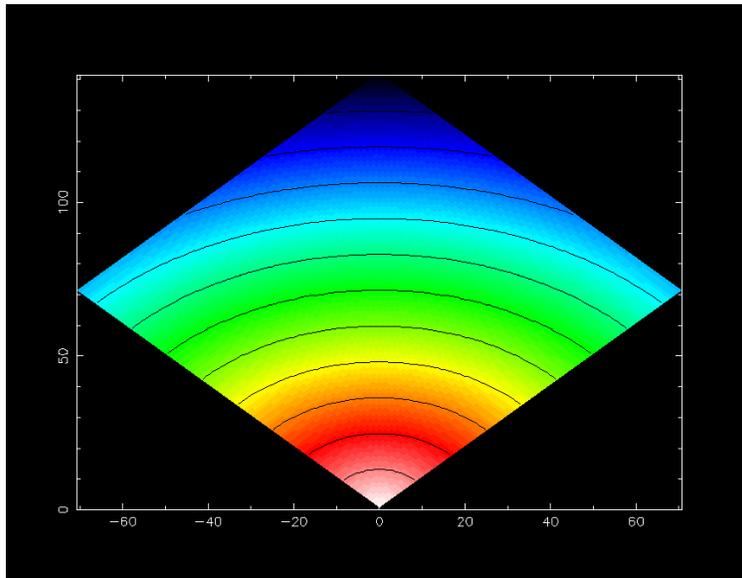


Figure 3: Graph produced by the example in section 7.3

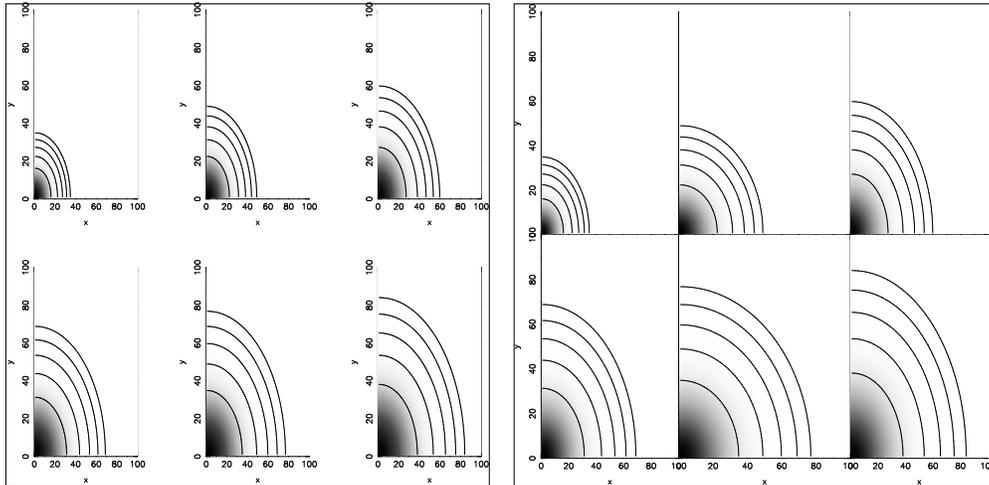


Figure 4: The `MPlot()` and `Panel()` graph layouts.

The `Panel` and `MPlot` have two mandatory arguments: `nx` and `ny`, the number of subplots in the x and y directions, respectively. There are two optional arguments `device` and `aspect`, which behave in the same way as the `Plot` class.

As an example, here is the python code which produced the plots shown in Figure 4. The only difference between the first and second of the graphs is the use of `Panel` instead of `Mplot`.

```
#!/usr/bin/env python
import pygplot
import Numeric
from math import *

# a function to create some 2-D data
sigma = 100.0
def gauss(xx,yy):
```

```

    global sigma
    sigma = float(sigma)
    return Numeric.exp(-(xx**2 + yy**2)/sigma)

# Now let's try some 2D data in a MPlot
# panel = pygplot.Mplot(3,2,device='test.ps/PS', aspect=1)
panel = pygplot.Panel(3,2,device='test.ps/PS', aspect=1)
for g in [100.0,200.0,300.0,400.0,500.0, 600.0]:
    sigma = g
    xy = Numeric.fromfunction(gauss, (100,100))

    pl = pygplot.Plot(xlabel='x', ylabel='y', ticks='in', fsize=2.0)
    pl.image(xy, palette='grey')
    pl.contour(xy, color=1, contours = [0.1, 0.01, 0.001, 0.0001, 0.00001])
    panel.add(pl)

panel.plot()
panel.close()

```

## 9 Interactive Sessions

One of the major incentives for  $\pi$ -gplot was to make something that would allow interactive plotting. The basic philosophy in designing the object oriented interface was to keep all the objects' attributes as member variables. In this way, one could change attributes such as `font` or `linestyle` and then `replot`. In this way, one can iterate on the displayed graph until satisfied, then produce a hardcopy by changing the `device` attribute of the top-level class.

Each object in  $\pi$ -gplot has a member variable called `options`. This is a dictionary holding the PGPLOT attributes that affect the object's appearance. Therefore, one can get a quick overview of the current state of an object's attributes by printing out the options:

```

>>> import pygplot
>>> plot = pygplot.Plot()
>>> plot.options
{'x2label': None, 'y2label': None, 'y2range': None, 'color': 'white', 'xrange':
None, 'symbol': 1, 'ylog': 0, 'y2log': 0, 'aspect': None, 'device': '/XWIN', 'fo
nt': 1, 'linestyle': 1, 'id': 0, 'x2range': None, 'linewidth': 1, 'yrange': None
, 'title': None, 'ticks': 'in', 'fsize': 1.0, 'yaxis': 0, 'width': 2, 'xlabel':
None, 'xaxis': 0, 'ylabel': None, 'nlyl': 0, 'x2log': 0, 'yformat': 'dec', 'nlxl
': 0, 'xlog': 0, 'xformat': 'dec'}
>>> plot.x2label = 'something new'
>>> plot.options
{'x2label': 'something new', 'y2label': None, 'y2range': None, 'color': 'white',
'xrange': None, 'symbol': 1, 'ylog': 0, 'y2log': 0, 'aspect': None, 'device': '/
XWIN', 'font': 1, 'linestyle': 1, 'id': 0, 'x2range': None, 'linewidth': 1, 'yra
nge': None, 'title': None, 'ticks': 'in', 'fsize': 1.0, 'yaxis': 0, 'width': 2,
'xlabel': None, 'xaxis': 0, 'ylabel': None, 'nlyl': 0, 'x2log': 0, 'yformat': 'd
ec', 'nlxl': 0, 'xlog': 0, 'xformat': 'dec'}

```

When the user calls the `line`, `symbol`, `contour` or any of the other member functions that produce graph elements, an object is created behind the scenes. If you want to be able to alter these elements later on (say, change the symbol of a data set), then you need to keep a reference to that object, which is returned by the member functions. For example, say we want to plot data `x` and `y` with symbol 0, but later change our minds:

```
>>> import pygplot
>>> x = [1,2,3,4,5]
>>> y = [0,0,1,0,0]
>>> plot = pygplot.Plot()
>>> line1 = plot.line(x,y, symbol=0)
>>> plot.plot()
>>> # we don't like the results
>>> line1.symbol = 1
>>> plot.replot()
>>> # Ah, that looks better, make a hard copy
>>> plot.close()
>>> plot.device = 'out.gif/GIF'
>>> plot.plot()
>>> plot.close()
```

## 10 Miscellaneous Functions

There are several helper functions in  $\pi$ -gplot. They are listed here, along with a short synopsis of their function.

1. `columns(filename)`: Read in a file of tabulated data and return the data as an array: each row of the array corresponding to a column in the data file. It will recognize space-, comma-, tab- or semi-colon-delimited columns.
2. `Linear(low,high,n)`: Outputs a list of contours, `n` in total, equally spaced between `low` and `high`.
3. `list_palettes()`: Outputs a list of available palettes which can be used with images.
4. `pgspline(xp, yp, x, y, slope0, slope1)`: Based on a set of points `(xp,yp)`, `pgspline` performs a spline interpolation at the points `x`. The arguments `slope0` and `slope1` specify the slope of the spline at the endpoints.

## 11 Calling PGPLOT Directly

PGPLOT has a number of functions or features which are not covered by the object oriented wrapper. However, it is still possible to use the PGPLOT functions directly, through the `ppgplot` module. Because the PGPLOT library is not object-oriented, you need to “select” a currently open PGPLOT device. Any PGPLOT commands will then apply to that device (and sub-panel, if the device is sub-divided). Each `Plot`, `MPlot` and `Panel` instance has a `select()` function. By calling this function, you set that instance to be the selected device. In the case of the `Plot` instance, `select()` doesn't have any arguments. For the `Panel` and `MPlot` instances, it takes two optional arguments: the row and column number of the

sub-panel you wish to select. Lastly, because the device needs to be open, you need to call the instance's `plot()` function before you can call the `select()` function and subsequently any `ppgplot` functions. Here follows an example of using the PGPLOT functions directly. Also note that as of version 0.7, the `ppgplot` module is located below the `pygplot` module, so you need to import it using `import pygplot.ppgplot`.

## 11.1 Example

In this python session, we setup a  $2 \times 2$  multi-plot. Then we draw in some arrows using `ppgplot.pgarro()`. The results are shown in Figure 5.

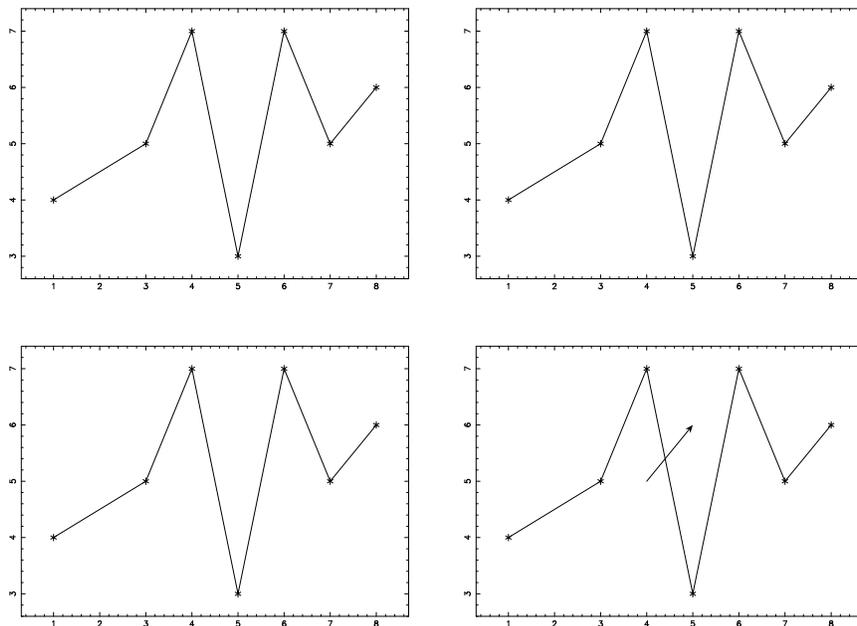


Figure 5: Graph produced by the example of section 11.1.

```
>>> import pygplot
>>> import pygplot.ppgplot as ppgplot
>>> import Numeric
>>> x = Numeric.array([1,3,4,5,6,7,8])
>>> y = Numeric.array([4,5,7,3,7,5,6])
>>> pl = pygplot.Plot()
>>> pl.point(x,y, symbol=3, size=2.0)
>>> pl.line(x,y, width=2)
>>> mp = pygplot.MPlot(2,2, device="arrow.ps/PS")
>>> for i in range(4):
>>>     mp.add(pl)
...
>>> mp.plot()
>>> mp.select(1,1)
>>> ppgplot.pgarro(4,5,5,6)
>>> mp.close()
```

## 12 API Documentation

I've used docstrings throughout the  $\pi$ -gplot code. This has two advantages: (1) you can use the `help()` function in interactive python sessions to get documentation (try `help(pygplot.Plot.line)` or `help(pygplot)`), and (2) I can automatically generate this documentation in HTML and L<sup>A</sup>T<sub>E</sub>X format. Along with this manual, you will find a file `api.tex`, which has all the function call summaries and brief explanations. Please report any omissions and/or inconsistencies. You can also find this API in HTML form in the `html` directory shipped with  $\pi$ -gplot and also online at <http://astro.swarthmore.edu/burns/pygplot>.