PyXPlot Users' Guide

A Commandline Plotting Package, with Interface similar to that of Gnuplot, which produces Publication-Quality Output.

Version 0.5.8

Dominic Ford
Trinity College
Cambridge
CB2 1TQ
UK
Email: dcf21@mrao.cam.ac.uk

September 2006

Contents

Chapter 1

Introduction

1.1 Overview

PyXPlot is a commandline graphing package, which, for ease of use, has an interface based heavily upon that of gnuplot – perhaps UNIX's most widely-used plotting package. Despite the shared interface, however, PyXPlot is intended to significantly improve upon the quality of gnuplot's output, producing publication-quality figures. The commandline interface has also been extended, providing a wealth of new features, and short-cuts for some operations which were felt to be excessively cumbersome in the original.

The motivation behind PyXPlot's creation was the apparent lack of a free plotting package which combined both high-quality output and a simple interface. Some – pgplot for one – provided very attractive output, but required a program to be written each time a plot was to be produced – a potentially time consuming task. Others, gnuplot being the prime example, were quick and simple to use, but produced less attractive results.

PyXPlot attempts to fill that gap, offering the best of both worlds. Though the interface is based upon that of gnuplot, text is now rendered with all of the beauty and flexibility of the LaTeX typesetting environment; the "multiplot" environment is made massively more flexible, making it easy to produce galleries of plots; and the range of possible output formats is extended – to name but a few of the enhancements. A number of examples of the results of which PyXPlot is capable can be seen in section 5.12.

As well as the ease of use and flexibility of gnuplot's commandline interface – it can be used either interactively, read a list of commands from a file, or receive instructions through a UNIX pipe from another process – I believe it to bring another distinct advantage. It insists upon data being written to a datafile on disk before being plotted. Packages which allow, or more often require, plotting to be done from within a programming language can encourage the calculation of data and its plotting to occur in the same program. I believe this to be a dangerous temptation, as the storage of raw datapoints to disk can then often be seen as a secondary priority. Months later, when the need arises to replot the same data in a different form, or to compare it with newer data, remembering how to use a hurriedly written program can prove tricky, but remembering how to plot a simple datafile less so.

The similarity of the interface to that of gnuplot is such that simple scripts written for gnuplot should work with PyXPlot with minimal modification; gnuplot users should be able to get started very quickly. However, as PyXPlot remains work in progress, it supports only a subset of the functionality and configurability of gnuplot, and some features may be found to be missing. These will be discussed further in section 2.9. A description of those features which have been added to the interface can be found in chapter 3.

A brief overview of gnuplot's interface is provided for novice users in chapter 2. However, the attention of past gnuplot users is drawn to one of the key changes to the interface – namely that all textual labels on plots are now printed using the LaTeX typesetting environment. This does unfortunately introduce some incompatibility with the original, since some strings which were valid before are no longer valid. For example:

```
set xlabel 'x^2'
```

would have been valid in gnuplot, but now needs to be written in LaTeX mathmode as:[1]

```
set xlabel '$x^2$'
```

It is the view of the author, however, that the nuisance of this incompatibility is far outweighed by the power that LaTeX brings. Users with no prior knowledge of LaTeX are advised that they don't know what they're missing, and that they should straight away download and read a copy of Tobias Oetiker's excellent introduction, *The Not So Short Guide to LaTeX2ε*[2].

[1] As in gnuplot, all textual labels in PyXPlot should be enclosed in either single or double quotes. If one were to want to render a string containing apostrophes, it would be necessary to enclose the string in double quotes, to prevent confusion between the apostrophe in the LaTeX, and the closing quote at the end of the line. However, to allow for those wanting to render LaTeX strings containing both single and double quote characters – for example, "`J\"org's Data`" – PyXPlot recognises the backslash character to be an escape character when followed by either ' or " in a LaTeX string. This is the *only* case in which PyXPlot considers \ an escape character. Consequently, in the example above, the "`\"`" would need to be double escaped: "`J\\"org's Data`".

[2] Download from:
`http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf`

1.2 System Requirements

PyXPlot is presently only supported for Linux. It requires that the following software packages (not included) be installed:

bash	(The bash shell)
python	(Version 2.4 or later)
scipy	(Python Scientific Library)
latex	(Used for all textual labels)
dvips	(Needed to render textual labels)
gs	(Ghostscript; needed for the landscape terminal)
convert	(ImageMagick; needed for the gif, png and jpg terminals)

The following package is not required for installation, but it is not possible to use the X11 terminal, i.e. to display plots on screen, without it:

gv	(Ghostview; used for the X11 terminal)

Debian users can find this software in the packages `tetex-extra`, `gv`, `imagemagick`, `python2.4`, `python2.4-scipy`.

1.3 Installation

The following steps describe the installation of PyXPlot from a `.tar.gz` archive. It is assumed that the packages listed above have already been installed.

- Unpack the distributed .tar.gz:

  ```
  tar xvfz pyxplot_0.5.7.tar.gz
  cd pyxplot
  ```

- Run the installation script:

  ```
  ./configure
  make
  make install
  ```

  where the final step needs to be executed as root. By default, the PyXPlot executable installs to `/usr/local/bin/pyxplot`. If desired, this installation path may be modified in the file `Makefile.skel`, by changing the variable `USRDIR` in the first line to an alternative desired installation location. This will be necessary for users who do not have root access to their machines, for example.

- Finally, start PyXPlot:

  ```
  pyxplot
  ```

1.4 Credits

Before proceeding any further, the author would like to express his gratitude to those people who have contributed to PyXPlot – first and foremost, to Jörg Lehmann and André Wobst, for writing the PyX graphics library for python, upon which this software is heavily built. Thanks must also go to Ross Church for his many useful comments and suggestions during its development.

1.5 Legal Blurb

Chapter 2

First Steps With PyXPlot

In this chapter, I shall provide a brief overview of the basic operation of PyXPlot, essentially covering those areas of syntax which are borrowed directly from gnuplot. Users who are already familiar with gnuplot may wish to skim or skip this chapter, though section 2.9, detailing which parts of gnuplot's interface are and are not supported in PyXPlot, may be of interest. In the following chapter, I shall go on to describe PyXPlot's extensions of gnuplot's interface.

Describing gnuplot's interface in its entirety is a substantial task, and what follows is only an overview; novice users can find many excellent tutorials on the web which will greatly supplement what is provided below.

2.1 Getting Started

The simplest way to start PyXPlot is simply to type "`pyxplot`" at a shell prompt to start an interactive session. A PyXPlot commandline prompt will appear, into which commands can be typed. PyXPlot can be exited either by typing "`exit`", "`quit`", or by pressing CTRL-D.

Alternatively, a list of commands to be executed may be stored in a command script, and executed by passing the filename of the command script to PyXPlot on the shell commandline, for example:

```
pyxplot foo
```

In this case, PyXPlot would exit immediately after finishing executing the commands from the file `foo`. Several filenames may be passed on the commandline, to be executed in sequence:

```
pyxplot foo1 foo2 foo3
```

Wildcards can also be used; the following would execute all command scripts in the presenting working directory whose filenames end with a `.plot` suffix:

```
pyxplot *.plot
```

It is possible to use PyXPlot both interactively, and from command scripts, in the same session. One way to do this is to pass the magic filename '–' on the commandline:

```
pyxplot foo1 - foo2
```

This magic filename represents an interactive session, which commences after the execution of `foo1`, and should be terminated in the usual way after use, with the "`exit`" or "`quit`" commands. Afterwards, the command script `foo2` would execute.

From within an interactive session, it is possible to run a command script using the `load` command:

```
pyxplot> load 'foo'
```

This example would have the same effect as typing the contents of the file `foo` into the present session.

A related command is "`save`", which stores a history of the commands executed in the present interactive session to file.

All command files can include comment lines, which should begin with a hash character, for example:

```
# This is a comment
```

Long commands may be split over multiple lines in the script by terminating each line of it with a backslash character, whereupon the following line will be appended to the end of it.

2.2 First Plots

The basic workhorse command of PyXPlot is the `plot` command, which is used to produce all plots. The following simple example would plot the function $\sin(x)$:

```
plot sin(x)
```

It is also possible to plot data from files. The following would plot data from a file '`datafile`', taking the x-coordinate of each point from the first column of the datafile, and the y-coordinate from the second. The datafile is assumed to be in plain text format, with columns separated by whitespace and/or commas[1]:

[1] If the filename of a datafile ends with a `.gz` suffix, it is assuming to be gzipped plaintext, and is decoded accordingly.

```
plot 'datafile'
```

Several items can be plotted on the same graph by separating them by commas:

```
plot 'datafile', sin(x), cos(x)
```

It is possible to define one's own variables and functions, and then plot them:

```
a = 2
b = 1
c = 1.5
f(x) = a*(x**2) + b*x + c
plot f(x)
```

To unset a variable or function once it has been set, the following syntax should be used:

```
a =
f(x) =
```

Labels can be applied to the two axes of the plot, and a title put at the top:

```
set xlabel 'This is the X axis'
set ylabel 'This is the Y axis'
set title 'A Plot of sin(x)'
plot sin(x)
```

All such text labels are displayed using LaTeX, and so any LaTeXcommands can be used, for example to put equations on axes:

```
set xlabel '$\frac{x^2}{c^2}$'
```

As a caveat, however, this does mean that care needs to be taken to escape any of LaTeX's reserved characters – i.e.: \ & % # { } $ _ ^ or ~.

Having set labels and titles, they may be removed thus:

```
set xlabel ''
set ylabel ''
set title ''
```

These are two other ways of removing the title from a plot:

```
set notitle
unset title
```

The `unset` command may be followed by essentially any word that can follow the `set` command, such as `xlabel` or `title`, to return that setting to its default configuration. The `reset` command restores all configurable parameters to their default states.

2.3 Plotting Datafiles

In the simple example of the previous section, we plotted the first column of a datafile against the second. It is also possible to plot any arbitrary column of a datafile against any other; the syntax for doing this is:

```
plot 'datafile' using 3:5
```

This example would plot the fifth column of the file `datafile` against the third. As mentioned above, columns in datafiles can be separated using whitespace and/or commas, which means that PyXPlot is compatible both with the format used by gnuplot, and also with comma-separated-value (CSV) files which many spreadsheets produce. Algebraic expressions may also be used in place of column numbers, for example:

```
plot 'datafile' using (3+$1+$2):(2+$3)
```

In algebraic expressions, column numbers should be prefixed by dollar signs, to distinguish them from numerical constants. The example above would plot the sum of the values in the first two columns of the datafile, plus three, on the horizontal axis, against two plus the value in the third column on the vertical axis. A more advanced example might be:

```
plot 'datafile' using 3.0:$($2)
```

This would place all of the datapoints on the line $x = 3$, drawing their vertical positions from the value of some column n in the datafile, where the value of n is itself read from the second column of the datafile.

Later, in section 3.3, I shall discuss how to plot rows of datafiles against one another, in horizontally arranged datafiles.

It is also possible to plot data from only a range of lines within a datafile. When PyXPlot reads a datafile, it looks for any blank lines in the file. It divides the datafile up into "data blocks", each being separated by single blank lines. The first datablock is numbered 0, the next 1, and so on.

When two or more blank lines are found together, the datafile is divided up into "index blocks". Each index block may be made up of a series of data blocks. To clarify this, a labelled example datafile is shown in figure 2.1.

By default, when a datafile is plotted, all data blocks in all index blocks are plotted. To plot only the data from one index block, the following syntax may be used:

```
plot 'datafile' index 1
```

To achieve the default behaviour of plotting all index blocks, the `index` modifier should be followed by a negative number.

`0.0`	`0.0`	Start of index 0, data block 0.
`1.0`	`1.0`	
`2.0`	`2.0`	
`3.0`	`3.0`	
		A single blank line marks the start of a new data block.
`0.0`	`5.0`	Start of index 0, data block 1.
`1.0`	`4.0`	
`2.0`	`2.0`	
		A double blank line marks the start of a new index.
		...
`0.0`	`1.0`	Start of index 1, data block 0.
`1.0`	`1.0`	
		A single blank line marks the start of a new data block.
`0.0`	`5.0`	Start of index 1, data block 1.
		<etc>

Figure 2.1: An Example PyXPlot Datafile – the datafile is shown in the two left-hand columns, and commands are shown to the right.

It is also possible to specify which lines and/or data blocks to plot from within each index. For this purpose the `every` modifier is used, which takes six values, separated by colons:

```
plot 'datafile' every a:b:c:d:e:f
```

The values have the following meanings:

a	Plot data only from every a th line in datafile.
b	Plot only data from every b th block within each index block.
c	Plot only from line c onwards within each block.
d	Plot only data from block d onwards within each index block.
e	Plot only up to the e th line within each block.
f	Plot only up to the f th block within each index block.

Any or all of these values can be omitted, and so the following would both be valid statements:

```
plot 'datafile' index 1 every 2:3
plot 'datafile' index 1 every :::3
```

The first would plot only every other data point from every third data block; the second from the third line onwards within each data block.

A final modifier for selecting which parts of a datafile are plotted is `select`, which plots only those data points which satisfy some given criterion. This is an extension of gnuplot's original syntax, and is described in section 3.3.

2.4 Directing Where Output Goes

By default, when PyXPlot is used interactively, all plots are displayed on the screen. It is also possible to produce postscript output, to be read into other programs or embedded into LaTeX documents, as well as a variety of other graphic formats. The `set terminal` command is used to specify the output format that is required, and the `set output` command the file to which output should be directed. For example,

```
set terminal postscript
set output 'myplot.eps'
plot sin(x)
```

would produce a postscript plot of $\sin(x)$ to the file `myplot.eps`.

The `set terminal` command can also be used to configure further aspects of the output file format. For example, the following would produce black-and-white and colour output respectively:

```
set terminal monochrome
set terminal colour
```

The former is useful for preparing plots for black-and-white publications, the latter for preparing plots for colourful presentations.

Both encapsulated and non-encapsulated postscript can be produced. Following gnuplot's slightly bizarre syntax, the word `enhanced` is used to produce encapsulated postscript, and `noenhanced` to produce normal postscript. The former is recommended for producing figures to embed into documents, the latter for plots which are to be printed without further processing:

```
set terminal noenhanced
set terminal enhanced
```

It is also possible to produce plots in the gif, png and jpeg graphic formats, as follows:

```
set terminal gif
set terminal png
set terminal jpg
```

More than one of the above keywords can be combined on a single line, for example:

```
set terminal postscript noenhanced colour
set terminal gif monochrome
```

To return to the default state of displaying plots on screen, the `x11` terminal should be selected:

```
set terminal x11
```

For more details of the `set terminal` command, including how to produce transparent gifs and pngs, see section 3.2.

We finally note that, after changing terminals, the `replot` command is especially useful; it repeats the last `plot` command.. If any plot items are placed after it, they are added to the last plot.

2.5 Data Styles

By default, data from files is plotted with points, and functions are plotted with lines. However, either kinds of data can be plotted in a variety of ways. To plot a function with points, for example, the following syntax is used[2]:

```
plot sin(x) with points
```

The number of points displayed (i.e. the number of samples of the function) can be set as follows:

```
set samples 100
```

Likewise, datafiles can be plotted with lines:

```
plot 'datafile' with lines
```

A variety of other styles are available. `linespoints` combines both the `points` and `lines` styles, drawing lines through points. Errorbars can also be drawn, as follows:

```
plot 'datafile' with yerrorbars
```

In this case, three columns of data need to be specified: the x- and y-coordinates of each datapoint, plus the size of the vertical errorbar on that datapoint. By default, the first three columns of the datafile are used, but once again (see section 2.3), the `using` modifier can be used:

```
plot 'datafile' using 2:3:7 with yerrorbars
```

[2]Note that when a plot command contains both `using`/`every` modifiers, and the `with` modifier, the latter must come last.

More details of the errorbars plot style can be found in section 3.3. Other plots styles supported by PyXPlot are listed in section 2.9, and their details can be found in many gnuplot tutorials. Bar charts will be discussed further in section 3.6.

The modifiers "`pointtype`" and "`linetype`", which can be abbreviated to "`pt`" and "`lt`" respectively, can also be placed after the `with` modifier. Each should be followed by an integer. The former specifies what shape of points should be used to plot the dataset, and the latter, whether a line should be continuous, dotted, dash-dotted, etc. Different integers correspond to different styles.

2.6 Setting Axis Ranges

In section 2.2, the `set xlabel` configuration command was previously introduced for placing text labels on axes. In this section, the configuration of axes is extended to setting their ranges.

By default, PyXPlot automatically scales axes to some sensible range which contains all of the plotted data. However, it is also possible for the user to override this and set his own range. This can be done directly from the plot command, for example:

```
plot [-1:1][-2:2] sin(x)
```

The ranges are specified immediately after the `plot` statement, with the syntax `[minimum:maximum]`.[3] The first specified range applies to the x-axis, and the second to the y-axis.[4] Any of the values supplied can be omitted, for example:

```
plot [:][-2:2] sin(x)
```

would only set a range on the y-axis.

Alternatively, ranges can be set before the `plot` statement, using the `set xrange` statement, for example:

```
set xrange [-2:2]
set y2range [a:b]
```

Having done so, a range may subsequently be turned off, and an axis returned to its default autoscaling behaviour, using the `set autoscale` command, which takes a list of axes to which it is to apply. If no list is supplied, then the command is applied to all axes.

[3] An alternative valid syntax is to replace the colon with the word 'to': `[minimum to maximum]`.

[4] As will be discussed in section 3.3.1, if further ranges are specified, they apply to the $x2$-axis, then the $y2$-axis, and so forth.

```
set autoscale x y
set autoscale
```

Axes can be set to have logarithmic scales using the `set logscale` command, which also takes a list of axes to which it should apply. Its converse is `set nologscale`:

```
set logscale
set nologscale y x x2
```

Further discussion of the configuration of axes can be found in section 3.3.1.

2.7 Function Fitting

It is possible to fit functional forms to data points in datafiles using the `fit` command. A simple example might be:

```
f(x) = a*x+b
fit f(x) 'datafile' index 1 using 2:3 via a,b
```

The coefficients to be varied are listed after the keyword "`via`"; the keywords `index`, `every` and `using` have the same meanings as in the plot command.[5]

This is useful for producing best-fit lines[6], and also has applications for estimating the gradients of datasets. The syntax is essentially identical to that used by gnuplot, though a few points are worth noting:

- When fitting a function of n variables, at least $n+1$ columns (or rows – see section 3.3) must be specified after the `using` modifier. By default, the first $n+1$ columns are used. These correspond to the values of each of the n inputs to the function, plus finally the value which the output from the function is aiming to match.

- If an additional column is specified, then this is taken to contain the standard error in the value that the output from the function is aiming to match, and can be used to weight the datapoints which are input into the `fit` command.

- By default, the starting values for each of the fitting parameters is 1.0. However, if the variables to be used in the fitting process are already set before the `fit` command is called, these initial values are used instead. For example, the following would use the initial values $\{a = 100, b = 50\}$:

[5] The `select` keyword, to be introduced in section 3.3 can also be used.

[6] Another way of producing best-fit lines is a to use a cubic spline; more details in given in section 3.8

```
f(x) = a*x+b
a = 100
b = 50
fit f(x) 'datafile' index 1 using 2:3 via a,b
```

- As with all numerical fitting procedures, the `fit` command comes with caveats. It uses a generic fitting algorithm, and may not work well with poorly behaved or ill-constrained problems. It works best when all of the values it is attempting to fit are of order unity. For example, in a problem where a was of order 10^{10}, the following might fail:

  ```
  f(x) = a*x
  fit f(x) 'datafile' via a
  ```

  However, better results might be achieved if a were artificially made of order unity, as in the following script:

  ```
  f(x) = 1e10*a*x
  fit f(x) 'datafile' via a
  ```

- A series of ranges may be specified after the `fit` command, using the same syntax as in the `plot` command, as described in section 2.6. If ranges are specified then only datapoints falling within these ranges are used in the fitting process; the ranges refer to each of the n variables of the fitted function in order.

- For those interested in the mathematical details, the workings of the `fit` command is discussed in more detail in chapter 6.

At the end of the fitting process, the best-fitting values of each parameter are output to the terminal, along with an estimate of the uncertainty in each. Additionally, the Hessian, covariance and correlation matrices are output in both human-readable and machine-readable formats, allowing a more complete assessment of the probability distribution of the parameters.

2.8 Interactive Help

In addition to this Users' Guide, PyXPlot also has a `help` command, which provides a hierarchical source of information. Typing 'help' alone gives a brief introduction to the help system, as well as a list of topics on which help is available. To display help on any given topic, type 'help' followed by the name of the topic. For example:

```
help commands
```

provides information on PyXPlot's commands. Some topics have subtopics, which are listed at the end of each page. To view them, add further words to the end of your help request – an example might be:

```
help commands help
```

which would display help on the `help` command itself.

2.9 Differences Between PyXPlot and Gnuplot

The commands supported by PyXPlot are only a subset of those available in gnuplot, although most of its functionality is present. Features which are supported by this version include:

- Allocation of user-defined variables and functions.
- The `print`, `help`, `exit` and `quit` commands.
- The `reset` and `clear` commands.
- The `!` command, to execute the remainder of the line as a shell command, e.g. `!ls`.
- The `cd` and `pwd` commands, to change and display the current working directory.
- The use of ‘ ‘ back-quotes to substitute the output of a shell command.[7]
- Set plot titles, axis labels, axis ranges, pointsize, linestyles, etc.
- Fitting of functions to data via the `fit` command.
- Basic 2d plotting and replotting of functions and datafiles, with the following styles: `lines`, `points`, `linespoints`, `dots`, `boxes`, `steps`, `fsteps`, `histeps`, `impulses`, `csplines`, `acsplines` and errorbars of all flavours (see section 3.3 for details of changes to errorbars).
- Automatic and manual selection of linestyles, linetypes, linewidths, pointtypes and pointsizes.
- Use of dual axes. Note: Operation here differs slightly from original gnuplot; dual axes are displayed whenever they are defined, there is no need to `set xtics nomirror`. See the details in the following section.
- Placing arrows and textual labels on plots.

[7] It should be noted that back-quotes can only be used outside quotes. For example, `set xlabel ’‘ls‘’` will not work. The best way to do this would be: `set xlabel ‘echo "’" ; ls ; echo "’"‘`.

- Putting grids on plots (colour can be set, but not linestyle).
- Setting plot aspect ratios with `set size ratio` or `set size square`.
- Multiplot (which is very significantly improved over gnuplot; see section 3.5)

Gnuplot features which PyXPlot does not presently support include:

- Parametric function plotting.
- Three-dimensional plotting (i.e. the `splot` command).
- Setting major/minor tics (but PyXPlot always gets this right without being told anyway ☺).[8]

[8] An effect similar to that of gnuplot's `set notics` command can be obtained with the magic `nolabelstics` axis label, described in section 3.3.1. The implementation of the `set tics` command is a high priority in version 0.6.x.

Chapter 3

Extensions of Gnuplot's Interface

A large number of new functions are available in PyXPlot which were not originally present in gnuplot. This chapter describes these extensions. From here onwards I shall presume that the user is familiar with the basic operation of gnuplot, and shall concentrate on the differences between PyXPlot's interface and that of gnuplot. In addition to having read the previous chapter, novice users may also find it of use to consult one of the many gnuplot tutorials which are to be found on the web before proceeding.

3.1 The Commandline Environment

PyXPlot uses the Gnu Readline commandline environment, which means that the up and down arrow keys can be used to repeat previously executed commands. Each user's command history is stored in his homespace in a history file called '`.pyxplot_history`', allowing PyXPlot to remember command histories between sessions. Additionally, a `save` command is provided, allowing the user to save his command history from the present session to a text file; this has the following syntax:

```
save 'output_filename'
```

From the shell commandline, the PyXPlot accepts the following switches which modify its behaviour:

`-h --help`	Display a short help message listing the available commandline switches.
`-v --version`	Display the current version number of PyXPlot.
`-q --quiet`	Turn off the display of the welcome message on startup.

`-V --verbose`	Display the welcome message on startup, as happens by default.
`-c --colour`	Use colour highlighting[1] to display output in green, warning messages in amber, and error messages in red.[2] These colours can be changed in the `terminal` section of the configuration file; see section 4.1 for more details.
`-m --monochrome`	Do not use colour highlighting, as happens by default.

3.2 Formatting and Terminals

In this section I shall outline the new and modified commands for controlling the graphic output format of PyXPlot.

The widths of plots may be set be means of two commands – `set size` and `set width`. Both are equivalent, and should be followed by the desired width measured in centimetres, for example:

```
set width 20
```

The `set size` command can also be used to set the aspect ratio of plots by following it with the keyword `ratio`. The number which follows should be the desired ratio of height to width. The following, for example would produce plots three times as high as they are wide:

```
set size ratio 3.0
```

The command `set size noratio` returns to PyXPlot's default aspect ratio of the golden ratio, i.e. $\left((1+\sqrt{5})/2\right)^{-1}$, which matches that of a sheet of A4 paper[3]. The special command `set size square` sets the aspect ratio to unity.

In section 2.4 I described how the `set terminal` command can be used to produce plots in various graphic formats. In addition, I here describe how the way in which plots are displayed on the screen can be changed. The default terminal, `X11`, is used to send output to screen.

By default, each time a new plot is generated, if the previous plot is still open on the display, the X11 terminal will replace it with the new one, thus keeping only one plot window open at a time. This has the advantage that the desktop does not become flooded with plot windows.

[1]This will only function on terminals which support colour output.

[2]The author apologies to those members of the population who are red/green colour-blind, but draws their attention to the following sentence.

[3]Of less practical significance, it has been in use since the time of the Pythagoreans, and is seen repeatedly in the architecture of the Parthenon.

If this behaviour is not desired, old plots can be kept visible when plotting further graphs by using the the `X11_multiwindow` terminal:

```
set terminal X11_singlewindow
plot sin(x)
plot cos(x)  <-- first plot window disappears
```

c.f.:

```
set terminal X11_multiwindow
plot sin(x)
plot cos(x)  <-- first plot window remains
```

As there are many changes to the options accepted by the `set terminal` command in comparison to those understood by gnuplot, the settings allowed in PyXPlot are listed below:

`x11_singlewindow`	Displays plots on the screen (in X11 windows, using ghostview). Each time a new plot is generated, it replaces the old one, preventing the desktop from becoming flooded with old plots.[4] **[default when running interactively; see below]**
`x11_multiwindow`	As above, but each new plot appears in a new window, and the old plots remain visible. As many plots as may be desired can be left on the desktop simultaneously.
`postscript`	Sends output to a postscript file. The filename for this file should be set using `set output`. **[default when running non-interactively; see below]**
`eps`	Equivalent to '`postscript enhanced`'.
`colour`	Allows datasets to be plotted in colour. Automatically they will be displayed in a series of different colours, or alternatively colours may be specified using the `with colour` plot modifier (see below). **[default]**
`color`	Equivalent to the above; provided for users of nationalities which can't spell. ☺
`monochrome`	Opposite to the above; all datasets will be plotted in black.
`enhanced`	Modifier for the postscript terminal; sets it to produce encapsulated postscript (eps) files. These can be embedded in documents, but do not print reliably.
`noenhanced`	Modifier for the postscript terminal; opposite to the above; sets it to produce printable postscript files.

[4] The author is aware of a bug, that this terminal can occasionally go blank when a new plot is generated. This is a known bug in ghostview, and can be worked around by selecting File → Reload within the ghostview window.

`portrait`	Sets plots to be displayed in upright (normal) orientation. [**default**]
`landscape`	Opposite of the above; produces side-ways plots. Not very useful when displayed on the screen, but you fit more on a sheet of paper that way around.
`gif`	Sends output to a gif image file; as above, the filename should be set using `set output`.
`png`	As above, but produces a png image.
`jpg`	As above, but produces a jpeg image.
`invert`	Modifier for the gif, png and jpg terminals; produces output with inverted colours.[5]
`noinvert`	Modifier for the gif, png and jpg terminals; opposite to the above. [**default**]
`transparent`	Modifier for the gif and png terminals; produces output with a transparent background.
`solid`	Modifier for the gif and png terminals; opposite to the above. [**default**]

The default terminal is normally `x11_singlewindow`, matching approximately the behaviour of gnuplot. However, there is an exception to this. When PyXPlot is used non-interactively – i.e. one or more command scripts is specified on the commandline, and PyXPlot exits as soon as it finishes executing them – the `x11_singlewindow` is not a very sensible terminal to use. Any plot window would close as soon as PyXPlot exited. The default terminal in this case changes to `postscript`.

One exception to this is when the special '–' filename is specified in a list of command scripts on the commandline, to produce an interactive terminal between running a series of scripts. In this case, PyXPlot detects that the session will be interactive, and defaults to the usual `x11_singlewindow` terminal.

An additional exception is on machines where the `DISPLAY` environment variable is not set. In this case, PyXPlot detects that it has access to no X-terminal on which to display plots, and defaults to the `postscript` terminal.

The `gif`, `png` and `jpg` terminals result in some loss of quality, since the plot has to be sampled into a bitmapped graphic format. By default, this sampling is performed at 300 dpi, though it may be changed using the command `set dpi <value>`. Alternatively, it may be changed using the `DPI` option in the `settings` section of a configuration file (see section 4.1).

[5]This terminal setting is useful for producing plots to embed in talk slideshows, which often contain bright text on a dark background. It only works when producing bitmapped output, though a similar effect can be achieved in postscript using the `set textcolour` and `set axescolour` commands (see section 3.4.3).

3.3 Plotting

In this section I outline some of the extensions of the `plot` command, to give greater flexibility in the appearance of graphs.

3.3.1 Configuring Axes

By default, plots have only one x-axis and one y-axis. Further parallel axes can be added and configured via statements such as:

```
set x3label 'foo'
plot sin(x) axes x3y1
set axis x3
```

In the top statement, a further x axis, called $x3$ is implicitly created by giving it a label. In the next, the `axes` modifier is used to tell the `plot` command to plot data against the $x3$-axis, which also implicitly created such an axis if it doesn't already exist. In the third, an $x3$-axis is explicitly created.

Unlike gnuplot, which allowed only a maximum of two parallel axes to be added to plots, PyXPlot allows an unlimited number of axes to be used. Odd-numbered x-axes appear below the plot, and even numbered x-axes above it; a similar rule applies for y-axes, to the left and to the right.

As discussed in the previous chapter, the ranges of axes can be set either using the `set xrange` command, or within the `plot` command. The following two statements would set an equivalent range for the $x3$-axis:

```
set x3range [-2:2]
plot [:][:][:][:][-2:2] sin(x) axes x3y1
```

As usual, the first two ranges specified in the `plot` command apply to the x- and y-axes. The next pair apply to the $x2$- and $y2$-axes, and so forth.

Having made axes with the above commands, they may subsequently be removed using the `unset axis` command as follows:

```
unset axis x3
unset axis x3x5y3 y7
```

The top statement, for example, would remove axis $x3$. The command `unset axis` on its own, with no axes specified, returns all axes to their default configuration. The special case of `unset axis x1` does not remove the first x-axis – it cannot be removed – but instead returns it to its default configuration.

It should be noted, that if the following two commands are typed in succession, the second may not entirely negate the first:

```
set x3label 'foo'
unset x3label 'foo'
```

The first may have implicitly created an $x3$-axis, which would need to be removed with the `unset axis x3` command.

A subtly different task is that of removing labels from axes, or setting axes not to display. To achieve this, a number of special axis labels are used. Labelling an axis "`nolabels`" has the effect that no title or numerical labels are placed upon it. Labelling it "`nolabelstics`" is stronger still; this removes all tick marks from it as well (similar in effect to `set noxtics` in gnuplot). Finally, labelling it "`invisible`" makes an axis completely invisible.

Labels may be placed on such axes, by following the magic keywords above with a colon and the desired title, for example:

```
set xlabel 'nolabels:Time'
```

would produce an x-axis with no numeric labels, but a label of 'Time'.

Several examples of effects which can be achieved with these commands can be found in Example 10 (see section 5.10). In the unlikely event of wanting to label a normal axis with one of these magic words, this may be achieved by prefixing the magic word with a space. There is one further magic axis label, `linkaxis`, which will be described in section 3.5.

The ticks of axes can be configured to point either inward, towards the plot, as is the default, or outward towards the axis labels, or in both directions. This is achieved using the `set xticdir` command, for example:

```
set xticdir inward
set y2ticdir outward
set x2ticdir both
```

3.3.2 Keys and Legends

By default, plots are displayed with a legend in their top-right corners. The textual description of each dataset is drawn by default from the command used to plot it. Alternatively, the user may specify his own description for each dataset by following the `plot` command with the `title` modifier, as follows:

```
plot sin(x) title 'A sine wave'
plot cos(x) title ''
```

In the lower case, a blank title is specified, in which case, PyXPlot makes no entry for this dataset in the legend. This is useful if it is desired to place some but not all datasets into the legend of a plot. Alternatively, the production of the legend can be completely turned off for all datasets, by the command `set nokey`. The opposite effect can be achieved by the `set key` command.

This latter command can also be used to dictate where on the plot the legend should be placed, using a syntax along the lines of:

```
set key top right
```

The following recognised positioning keywords are self-explanatory: `top`, `bottom`, `left`, `right`, `xcentre` and `ycentre`. The word `outside` places the key outside the plot, on its right side. The word `below` places the legend below the plot.

In addition, two positional offset coordinates may be specified after such keywords – the first value is assumed to be an x-offset, and the second a y-offset, in units approximately equal to the size of the plot. For example:

```
set key bottom left 0.0 -0.5
```

would display a key below the bottom left corner of the graph.

By default, entries in the key are placed in a single vertical list. They can instead be arranged into a number of columns by means of the `set keycolumns` command.. This should be followed by the integer number of desired columns, for example:

```
set keycolumns 2
```

3.3.3 The `linestyle` keyword

At times, the string of style keywords following the `with` modifier in `plot` commands can grow rather unwieldily long. For clarity, frequently used plot styles can be stored as "linestyles"; this is true of styles involving points as well as lines. The syntax for setting a linestyle is:

```
set linestyle 2 points pointtype 3
```

where the "2" is the identification number of the linestyle. In a subsequent `plot` statement, this linestyle can be recalled as follows:

```
plot sin(x) with linestyle 2
```

3.3.4 Colour Plotting

In the `with` clause of the plot command, the modifier `colour`, (abbrev. '`c`'), allows the colour of each dataset to be manually selected. It should be followed either by an integer, to set a colour from the present palette, or by a colour name. A list of valid colour names is given in section 4.6. For example:

```
plot sin(x) with c 5
plot sin(x) with colour blue
```

The `colour` modifier can also be used when defining linestyles.

PyXPlot has a palette of colours which it assigns sequentially to datasets when colours are not manually assigned. This is also the palette to which integers passed to `set colour` refer – the '5' above, for example. It may be set using the `set palette` command, which differs in syntax from gnuplot. It should be followed by a comma-separated list of colours, for example:

```
set palette red,green,blue
```

Another way of setting the palette, in a configuration file, is described in section 4.2; a list of valid colour names is given in section 4.6.

3.3.5 General Extensions Beyond Gnuplot

plot linewidths — For an unknown reason, gnuplot doesn't allow `set linewidth 2` as valid syntax. This setting is allowed to be made in PyXPlot. Furthermore, `set pointlinewidth 2` will set the linewidth to be used when drawing data *points*. A similar effect can be achieved via:

```
plot sin(x) with points pointlinewidth 2
```

In both cases, the abbreviation `plw` is valid.

dots plot style — When using the `dots` style, for example:

```
plot sin(x) with dots
```

the size of the plotted dots can be varied with the `pointsize` modifier, unlike in gnuplot, where the dots were of a fixed size. For example, to display big dots, use:

```
plot sin(x) with dots pointsize 10
```

`select` keyword — As well as the `index`, `using` and `every` keywords which gnuplot used to allow users to plot subsets of data from datafiles, PyXPlot also has a further modifier, `select`. This can be used to plot only those datapoints in a datafile which specify some given criterion. For example:

```
plot 'datafile' select ($8>5)
plot sin(x) select (($1>0)and($2>0))
plot sin(x) select ($1>0) select ($2>0)
```

In the third example, two select criteria are given; it is entirely equivalent to the statement above it. Note that whitespace is not permitted in `select` criteria. The select modifier has many applications, including plotting two-dimensional slices from three-dimensional datasets, and selecting certain subsets of datapoints from a datafile for plotting.

Logical operators such as `and`, `or` and `not` can be used, as seen in the second example above; indeed, any expression which is valid Python can be used.

arrows plot style — A new plotting style, `arrows`, is available, which takes four columns of data, x_1, y_1, x_2, y_2, and for each data point draws an arrow from the point (x_1, y_1) to (x_2, y_2). Three different kinds of arrows can be drawn: ones with normal arrow heads, ones with no arrow heads, which just appear as lines, and ones with arrow heads on both ends. The syntax is:

```
plot 'datafile' with arrows_head
plot 'datafile' with arrows_nohead
plot 'datafile' with arrows_twohead
```

The syntax '`with arrows`' is a shorthand for '`with arrows_head`'.

lower and upper limit datapoints — PyXPlot can plot datapoints using the standard upper- and lower-limit symbols. No special syntax is required for this; these symbols are pointtypes[6] 12 and 13 respectively, obtained as follows:

```
plot 'upperlimits' with points pointtype 12
plot 'lowerlimits' with points pointtype 13
```

[6] The `pointtype` modifier was introduced in section 2.5.

plotting functions with errorbars and other plot styles — In gnuplot, when a function (as opposed to a datafile) is plotted, only those plot styles which accept two columns of data can be used – for example, `lines` or `points`. It is not possible to plot a function with errorbars, for example. In PyXPlot, by contrast, this is possible using the following syntax:

```
plot f(x):g(x) with yerrorbars
```

Two functions are supplied, separated by a colon; plotting proceeds as if a datafile had been supplied, containing values of x in column 1, values of $f(x)$ in column 2, and values of $g(x)$ in column 3. This may be useful, for example, if $g(x)$ measures the intrinsic uncertainty in $f(x)$. The `using` modifier may also be used:

```
plot f(x):g(x) using 2:3
```

Here, $g(x)$ would be plotted on the y-axis, against $f(x)$ on the x-axis. It should be noted, however, that the range of values of x used would still correspond to the range of the plot's horizontal axis. If the above were to be attempted with an autoscaling horizontal axis, the result might be rather unexpected – PyXPlot would find itself autoscaling the x-axis range to the spread of values of $f(x)$, but find that this itself changed depending upon the range of the x-axis.

horizontally arranged datafiles — The command syntax for plotting columns of datafiles against one another was previously described in section 2.3. In an extension of gnuplot's interface, it is also possible to plot *rows* of data against one another in horizontally-arranged datafiles. For this, the keyword '`rows`' is placed after the `using` modifier:

```
plot 'datafile' index 1 using rows 1:2
```

The syntax '`using columns`' is also accepted, to specify the default behaviour of plotting columns against one another:

```
plot 'datafile' index 1 using columns 1:2
```

When plotting horizontally-arranged datafiles, the meanings of the `index` and `every` modifiers (see section 2.3) are altered slightly. The former continues to refer to vertical blocks of data separated by two blank lines. Blocks, as referenced in the `every` modifier, continue to be vertical blocks of datapoints, separated by single blank lines. The row numbers passed to the `using` modifier are counted from the top of the current block.
However, the line-numbers specified in the `every` modifier – i.e. variables a, c and e in the system above – now refer to horizontal columns, rather than lines. For example:

```
plot 'datafile' using rows 1:2 every 2::3::9
```

would plot the data in row 2 against that in row 1, using only the values in every other column, between columns 3 and 9.

errorbars — In gnuplot, when one used errorbars, one could either specify the size of the errorbar, or the min/max range of the errorbar. Both of these usages shared a common syntax, and gnuplot's behaviour depended upon the number of data columns provided:

```
plot 'datafile' with yerrorbars
```

Given a datafile with three columns, this would take the third column to indicate the size of the y-errorbar, and given a four-column datafile, it would take the third and fourth columns to indicate the min/max range to be marked out by the errorbar.
To avoid confusion, a different syntax is adopted in PyXPlot. The syntax:

```
plot 'datafile' with yerrorbars
```

now always assumes the third column of the datafile to indicate the size of the errorbar, regardless of whether a fourth is present. The syntax:

```
plot 'datafile' with yerrorrange
```

always assumes the third and fourth columns to indicate the min/max range of the errorbar.

For clarity, a complete list of errorbar styles is given below:

`yerrorbars`	Vertical errorbars; size drawn from the third data-column.
`xerrorbars`	Horizontal errorbars; size drawn from the third data-column.
`xyerrorbars`	Horizontal and vertical errorbars; sizes drawn from the third and fourth data-columns respectively.
`errorbars`	Shorthand for `yerrorbars`.
`yerrorrange`	Vertical errorbars; minimum drawn from the third data-column, maximum from the fourth.
`xerrorrange`	Horizontal errorbars; minimum drawn from the third data-column, maximum from the fourth.
`xyerrorrange`	Horizontal and vertical errorbars; horizontal minimum drawn from the third data-column, and maximum from the fourth; vertical minimum drawn from the fifth, and maximum from the sixth.
`errorrange`	Shorthand for `yerrorrange`.

datafile wildcards — PyXPlot allows the wildcards '*' and '?' to be used both in the filenames of datafiles following the `plot` command, and also when specifying command files on the commandline and with the `load` command. For example, the following would plot all datafiles in the current directory with a '.dat' suffix, using the same plot options:

```
plot '*.dat' with linewidth 2
```

In the legend, full filenames are displayed, allowing the datafiles to be distinguished.

As in gnuplot, a blank filename passed to the plot command causes the last used datafile to be used again.

backing up overwritten files — By default, when plotting to a file, if the output filename matches that of an existing file, that file is overwritten. This behaviour may be changed with the `set backup` command, which has syntax:

```
set backup
set nobackup
```

When this switch is turned on, pre-existing files will be renamed with a tilda at the end of their filenames, rather than being overwritten.

3.4 Sundry Items (Arrows, Text Labels, and More)

This section describes how to put arrows and text labels on plots; the syntax is similar to that used by gnuplot, but slightly changed. It is now possible, for example, to set the linestyles and colours with which arrows should be drawn. Also covered is how to put grids onto plots, and how to change the size and colour of textual labels on plots.

3.4.1 Arrows

Arrows may be placed on plots using the `set arrow` command, which has similar syntax to that used by gnuplot. A simple example would be:

```
set arrow 1 from 0,0 to 1,1
```

The number '1' immediately following 'set arrow' specifies an identification number for the arrow, allowing it to be subsequently removed via:

```
unset arrow 1
```

or equivalently, via:

```
set noarrow 1
```

In PyXPlot, this syntax is extended; the `set arrow` command can be followed by the keyword '`with`', to specify the style of the arrow. For example, the specifiers '`nohead`', '`head`' and '`twohead`', after the keyword '`with`', can be used to make arrows with no arrow heads, normal arrow heads, or two arrow heads. '`twoway`' is an alias for '`twohead`'. For example:

```
set arrow 1 from 0,0 to 1,1 with nohead
```

In addition, linestyles and colours can be specified after the keyword '`with`':

```
set arrow 1 from 0,0 to 1,1 with nohead \
linetype 1 c blue
```

As in gnuplot, the coordinates for the start and end points of the arrow can be specified in a range of coordinate systems. '`first`', the default, measures the graph using the x- and y-axes. '`second`' uses the $x2$- and $y2$-axes. '`screen`' and '`graph`' both measure in centimetres from the origin of the graph. In the following example, we use these specifiers, and specify coordinates using variables rather than doing so explicitly:

```
x0 = 0.0
y0 = 0.0
x1 = 1.0
y1 = 1.0
set arrow 1 from first  x0, first  x1 \
            to   screen x1, screen x1 \
            with nohead
```

In addition to these four options, which are those available in gnuplot, the syntax '`axis`n' may also be used, to use the nth x- or y-axis – for example, '`axis3`'. This allows arrows to reference any arbitrary axis on plots which make use of large numbers of parallel axes (see section 3.3.1).

3.4.2 Text Labels

Text labels may be placed on plots using the `set label` command. As with all textual labels in PyXPlot, these are rendered in LaTeX:

```
set label 1 'Hello World' at 0,0
```

As in the previous section, the number '1' is a reference number, which allows the label to be removed by either of the following two commands:

```
set nolabel 1
unset label 1
```

The positional coordinates for the text label, placed after the keyword '`at`', can be specified in any of the coordinate systems described for arrows above.

The fontsize of these text labels can globally be set using the `set fontsize x` command. This applies not only to the `set label` command, but also to plot titles, axis labels, keys, etc. The value given should be an integer in the range $-4 \leq x \leq 5$. The default is zero, which corresponds to LaTeX's `normalsize`; -4 corresponds to `tiny` and 5 to `Huge`.

The `set textcolour` command can be used to globally set the colour of all text output, and applies to all of the text that the `set fontsize` command does. It is especially useful when producing plots to be embedded in presentation slideshows, where bright text on a dark background may be desired. It should be followed either by an integer, to set a colour from the present palette, or by a colour name. A list of the recognised colour names can be found in section 4.6. For example:

```
set textcolour 2
set textcolour blue
```

By default, each label's specified position corresponds to its bottom left corner. This alignment may be changed with the `set texthalign` and `set textvalign` commands. The former takes the options `left`, `centre` or `right`, and the latter takes the options `bottom`, `centre` or `top`, for example:

```
set texthalign right
set textvalign top
```

3.4.3 Gridlines

Gridlines may be placed on a plot and subsequently removed via the statements:

```
set grid
set nogrid
```

respectively. The following commands are also valid:

```
unset grid
unset nogrid
```

By default, gridlines are drawn from the major and minor ticks of the x- and y-axes. However, the axes which should be used may be specified after the `set grid` command:

```
set grid x2y2
set grid x x2y2
```

The top example would connect the gridlines to the ticks of the $x2$- and $y2$- axes, whilst the lower would draw gridlines from both the x- and the $x2$-axes.

If one of the specified axes does not exist, then no gridlines will be drawn in that direction. Gridlines can subsequently be removed selectively from some axes via:

```
unset grid x2x3
```

The colours of gridlines can be controlled via the `set gridmajcolour` and `set gridmincolour` commands, which control the gridlines emanating from major and minor axis ticks respectively. An example would be:

```
set gridmincolour blue
```

Any of the colour names listed in section 4.6 can be used.

A related command is `set axescolour`, which has a syntax similar to that above, and sets the colour of the graph's axes.

3.5 Multi-plotting

Gnuplot has a plotting mode called "multiplot" which allows many graphs to be plotted together, and display side-by-side. The basic syntax of this mode is reproduced in PyXPlot, but is hugely extended.

The mode is entered by the command "`set multiplot`". This can be compared to taking a blank sheet of paper on which to place plots. Plots are then placed on that sheet of paper, as usual, with the `plot` command. The position of each plot is set using the `set origin` command, which takes a comma-separated x, y coordinate pair, measured in centimetres. The following, for example, would plot a graph of $\sin(x)$ to the left of a plot of $\cos(x)$:

```
set multiplot
plot sin(x)
set origin 10,0
plot cos(x)
```

The multiplot page may subsequently be cleared with the `clear` command, and multiplot mode may be left using the "`set nomultiplot`" command.

At this point we move beyond the syntax available in gnuplot. Each time a plot is placed on the multiplot page in PyXPlot, it is allocated a reference number, which is output to the terminal. Reference numbers count up from zero each time the multiplot page is cleared. A number of commands exist for modifying plots after they have been placed on the page, selecting them by making reference to their reference numbers.

Plots may be removed from the page with the `delete` command, and restored with the `undelete` command:

```
delete <number>
undelete <number>
```

The reference numbers of deleted plots are not reused until the page is cleared, as they may always be restored with the `undelete` command; plots which have been deleted simply do not appear.

Plots may also be moved with the `move` command. For example, the following would move plot 23 to position (8,8) measured in centimetres:

```
move 23 8,8
```

The axes of plots can be linked together, in such a way that they always share a common scale. This can be useful when placing plots next to one another, firstly, of course, if it is of intrinsic interest to ensure that they are on a common scale, but also because the two plots then do not both need their own axis labels, and space can be saved by one sharing the labels from the other. In PyXPlot, an axis which borrows its scale and labels from another is called a "linked axis".

Such axes are declared by setting the label of the linked axis to a magic string such as "`linkaxis 0`". This magic label would set the axis to borrow its scale from an axis from plot zero. The general syntax is "`linkaxis` n m", where n and m are two integers, separated by a comma or whitespace. The first, n, indicates the plot from which to borrow an axis; the second, m, indicates whether to borrow the scale of axis $x1$, $x2$, $x3$, etc. By default, $m = 1$. The linking will fail, and a warning result, if an attempt is made to link to an axis which doesn't exist.

The specimen plots in section 5.12 show numerous examples of the use of linked axes.

In multiplot mode, the `replot` command can be used to modify the last plot added to the page. For example, the following would change the title of the latest plot to "foo", and add a plot of $\cos(x)$ to it:

```
set title 'foo'
replot cos(x)
```

Additionally, it is possible to modify any plot on the page, by first selecting it with the `edit` command. Subsequently, the `replot` will act upon the selected plot. The following example would produce two plots, and then change the colour of the text on the first:

```
set multiplot
plot sin(x)
set origin 10,0
plot cos(x)
edit 0        # Select the first plot ...
set textcolour red
replot        # ... and replot it.
```

The `edit` command can also be used to view the settings which are applied to any plot on the multiplot page – after executing "edit 0", the `show` command will show the settings applied to plot zero.

When a new plot is added to the page, `replot` always switches to act upon this most recent plot.

In addition to placing plots on the multiplot page, text labels may also be inserted independently of any plots, using the `text` command. This has the following syntax:

```
text 'This is some text' x,y
```

In this case, the string "This is some text" would be rendered at position (x, y) on the multiplot. The commands `set textcolour`, `set texthalign` and `set textvalign`, which have already been described in the context in the `set label` command, can also be used to set the colour and alignment of text produced with the `text` command.. A useful application of this is to produce centred headings at the top of multiplots.

As with plots, each text item has a unique identification number, and can be moved around, deleted or undeleted:

```
delete_text <number>
undelete_text <number>
move_text <number> x,y
```

It should be noted that the `text` command can also be used outside of the multiplot environment, to render a single piece of short text instead of a graph. This has limited applications, but one is illustrated in section 5.4.

Arrows may also be placed on multiplot pages, independently of any plots, using the `arrow` command, which has syntax:

```
arrow from x,y to x,y
```

As above, arrows receive unique identification numbers, and can be deleted and undeleted, though they cannot be moved:

```
delete_arrow <number>
undelete_arrow <number>
```

The `arrow` command may be followed by the '`with`' keyword to specify to style of the arrow. The style keywords which are accepted are identical to those accepted by the `set arrow` command (see section 3.4.1). For example:

```
arrow from x1,y1 to x2,y2 \
with twohead colour red
```

The `refresh` command is rather similar to the `replot` command, but produces an exact copy of the latest display. This can be useful, for example, after changing the terminal type, to produce a second copy of a multiplot page in a different format. But the crucial difference between this command and `replot` is that it doesn't replot anything. Indeed, there could be only textual items and arrows on the present multiplot page, and no graphs *to* replot.

3.5.1 Speed Issues

By default, whenever an item is added to a multiplot, or an existing item moved or replotted, the whole multiplot is replotted to show the change. This can be a time consuming process on large and complex multiplots. For this reason, the `set nodisplay` command is provided, which stops PyXPlot from producing any output. The `set display` command can subsequently be issued to return to normal behaviour.

This can be especially useful in scripts which produce large multiplots. There is no point in producing output at each step in the construction of a large multiplot, and so a great speed increase can be achieved by wrapping the script with:

```
set nodisplay
[...prepare large multiplot...]
set display
refresh
```

The reader will observe that frequent use of this is made in the examples of chapter 5.

3.6 Barcharts and Histograms

3.6.1 Basic Operation

As in gnuplot, bar charts and histograms can be produced using the `boxes` plot style:

```
plot 'datafile' with boxes
```

Horizontally, the interfaces between the bars are, by default, at the mid-points along the x-axis between the specified datapoints (see, for example, panel (a) of figure 5.7, and the script which produced it, in section 5.7). Alternatively, the widths of the bars may be set using the `set boxwidth` command. In this case, all of the bars will be centred upon their specified x-coordinates, and have total widths equalling that specified in the `set` boxwidth command. Consequently, there may be gaps between them, or they may overlap, as seen in panel (c) of figure 5.7.

Having set a fixed box width, the default automatic width mode may be restored either with the unset boxwidth command, or by setting the boxwidth to a negative width.

As a third alternative, it is also possible to specify different widths for each bar manually, in a column of the input datafile. For this, the wboxes plot style should be used:

```
plot 'datafile' using 1:2:3 with wboxes
```

This plot style expects three columns of data to be specified: the x- and y-coordinates of each bar, and the width in the third column. Panel (b) of figure 5.7 shows an example of this plot style in use.

By default, the bars all originate from the line $y = 0$, as is normally wanted for a histogram. However, should it be desired for the bars to start from a different vertical point, that may be achieved with the set boxfrom command, for example:

```
set boxfrom 5
```

All of the bars would then originate from the line $y = 5$. Panel (f) of figure 5.6 shows the kind of effect that is achieved; for comparison, panel (b) of the same figure shows the same bar chart with the boxes starting from their default position at $y = 0$.

The bars may be filled using the with fillcolour modifier, followed by the name of a colour:

```
plot 'datafile' with boxes fillcolour blue
plot 'datafile' with boxes fc 4
```

Additionally, the word 'auto' may be used in place of a colour name, to fill the bar with the line colour being used to draw it. Panels (c) and (d) of figure 5.7 demonstrate the use of filled bars.

Finally, the impulses plot style, as in gnuplot, produces bars of zero width; see panel (e) of figure 5.6 for an example.

3.6.2 Stacked Bar Charts

If several datapoints are supplied at a common x-coordinate to the boxes or wboxes plot styles, then the bars are stacked one above another into a stacked barchart. Consider the following datafile:

```
1 1
2 2
2 3
3 4
```

The second bar at $x = 2$ would be placed on top of the first, spanning the range $2 < y < 5$, and having the same width as the first. If plot colours are being automatically selected from the palette, then a different palette colour is used to plot the upper bar.

3.6.3 Steps

As an alternative to solid boxes, a graph may also be plotted with "steps"; see panels (a), (c) and (d) of figure 5.6 for examples. As is illustrated by these panels, three flavours of steps are available (exactly as in gnuplot):

```
plot 'datafile' with steps
plot 'datafile' with fsteps
plot 'datafile' with histeps
```

When using the `steps` plot style, the datapoints specify the right-most edges of each step. By contrast, they specify the left-most edges of the steps when using the `fsteps` plot style. The `histeps` plot style works rather like the `boxes` plot style; the interfaces between the steps occur at the horizontal midpoints between the datapoints.

3.7 Function Splicing

In PyXPlot, as in gnuplot, user-defined functions may be declared on the commandline:

```
f(x) = x*sin(x)
```

As an extension to what is possible in gnuplot, it is also possible to declare functions which are only valid over a certain range of argument space. For example, the following function would only be valid in the range $-2 < x < 2$:[7]

```
f(x)[-2:2] = x*sin(x)
```

The following function would only be valid when all of a, b, c were in the range $-1 \rightarrow 1$:

```
f(a,b,c)[-1:1][-1:1][-1:1] = a+b+c
```

If an attempt is made to evaluate a function outside of its specified range, then an error results. This may be useful, for example, for plotting a function, but not continuing it outside some specified range. The following would print the function $\sin(x)$, but only in the range $-2 < x < 7$:

[7] The syntax `[-2:2]` can also be written `[-2 to 2]`.

```
f(x)[-2:7] = sin(x)
plot f(x)
```

The output of this particular example can be seen in panel (a) of figure 5.9. A similar effect could also have been achieved with the `select` keyword; see section 3.3.

It is possible to make multiple declarations of the same function, over different regions of argument space; if there is an overlap in the valid argument space for multiple definitions, then later declarations take precedence. This makes it possible to use different functional forms for a function in different parts of parameter space, and is especially useful when fitting a function to data, if different functional forms are to be spliced together to fit different regimes in the data.

Another application of function splicing is to work with functions which do not have analytic forms, or which are, by definition, discontinuous, such as top-hat functions or Heaviside functions. The following example would define $f(x)$ to be a Heaviside function:

```
f(x) = 0
f(x)[0:] = 1
```

The declaration of a function similar to a top-hat function is demonstrated in panel (b) of figure 5.9. The following example would define $f(x)$ to follow the Fibonacci sequence, though it is not at all computationally efficient, and it is inadvisable to evaluate it for $x > 8$:

```
f(x) = 1
f(x)[2:] = f(x-1) + f(x-2)
plot [0:8] f(x)
```

3.8 Datafile Interpolation: Spline Fitting

Gnuplot allows data to be interpolated using its `csplines` plot style, for example:

```
plot 'datafile' with smooth csplines
plot 'datafile' with smooth acsplines
```

where the upper statement fits a spline through all of the datapoints, and the lower applies some smoothing to the data first. This syntax is supported in PyXPlot but deprecated. A similar effect can be achieved with the new, more powerful, `spline` command. This has a syntax similar to that of the `fit` command, for example:

```
spline f() 'datafile' index 1 using 2:3
```

The function $f(x)$ now becomes a special function, representing a spline fit to the given datafile. It can be plotted or otherwise used in exactly the same way as any other function. This approach is more flexible than gnuplot's syntax, as the spline $f(x)$ can subsequently be spliced together with other functions (see the previous section), or used in any mathematical operation. The following code snippet, for example, would fit splines through two datasets, and then plot the interpolated differences between them, regardless, for example, of whether the two datasets were sampled at exactly the same x coordinates:

```
spline f() 'datafile1'
spline g() 'datafile2'
plot f(x)-g(x)
```

Smoothed splines can also be produced:

```
spline f() 'datafile1' smooth 1.0
```

where the value 1.0 determines the degree of smoothing to apply; the higher the value, the more smoothing is applied. The default behaviour is not to smooth at all (equivalent to `smooth 0.0`); a value of 1.0 corresponds to the default amount of smoothing applied in the `acsplines` plot style.

3.9 Numerical Integration and Differentiation

Special functions are available for performing numerical integration and differentiation of expressions: `int_dx()` and `diff_dx()`. In each case, the "`x`" may be replaced with any valid variable name, to integrate or differentiate with respect to any given variable.

The function `int_dx()` takes three parameters – firstly the expression to be integrated, followed by the minimum and maximum integration limits. For example, the following would plot the integral of the function $\sin(x)$:

```
plot int_dt(sin(t),0,x)
```

The function `diff_dx()` takes two parameters and an optional third – firstly the expression to be differentiated, then the point at which the differential should be evaluated, and then an optional parameter, ϵ. The following example would evaluate the differential of the function $\cos(x)$ with respect to x at $x = 1.0$:

```
print diff_dx(cos(x), 1.0)
```

Differentials are evaluated by a simple differencing algorithm, and the parameter ϵ controls the spacing with which to perform the differencing operation:

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=x_0} \approx \frac{f(x_0+\epsilon/2)-f(x_0-\epsilon/2)}{\epsilon}$$

By default, $\epsilon = 10^{-6}$.

Advanced users may be interested to know that integration is performed using the `quad` function of the `integrate` package of the `scipy` numerical toolkit for Python – a general purpose integration routine.

3.10 Script Watching: pyxplot_watch

PyXPlot includes a simple tool for watching command script files, and executing them whenever they are modified. This may be useful when developing a command script, if one wants to make small modifications to it, and see the results in a semi-live fashion. This tool is invoked by calling the `pyxplot_watch` command from a shell prompt. The commandline syntax of `pyxplot_watch` is similar to that of PyXPlot itself, for example:

```
pyxplot_watch script
```

would set `pyxplot_watch` to watch the command script file `script`. One difference, however, is that if multiple script files are specified on the commandline, they are watched and executing independently, *not* sequentially, as PyXPlot itself would do. Wildcard characters can also be used to set `pyxplot_watch` to watch multiple files.[8]

This is especially useful when combined with GhostView's watch facility. For example, suppose that a script `foo` produces postscript output `foo.ps`. The following two commands could be used to give a live view of the result of executing this script:

```
gv --watch foo.ps &
pyxplot_watch foo
```

[8]Note that `pyxplot_watch *.script` and `pyxplot_watch \*.script` will behave differently in most UNIX shells. In the first case, the wildcard is expanded by your shell, and a list of files passed to `pyxplot_watch`. Any files matching the wildcard, created after running `pyxplot_watch`, will not be picked up. In the latter case, the wildcard is expanded by `pyxplot_watch` itself, which *will* pick up any newly created files.

Chapter 4

Configuring PyXPlot

4.1 Overview

As is the case in gnuplot, PyXPlot can be configured using the `set` command – for example:

```
set output 'foo.eps'
```

would set it to send its plotted output to the file `foo.eps`. Typing '`set`' on its own returns a list of all recognised '`set`' configuration parameters. The `unset` command may be used to return settings to their default values; it recognises a similar set of parameter names, and once again, typing '`unset`' on its own gives a list of them. The `show` command can be used to display the values of settings.

4.2 Configuration Files

PyXPlot can also be configured by means of a configuration file, with filename `.pyxplotrc`, which is scanned once upon startup. This file may be placed either in the user's current working directory, or in his home directory. In the event of both files existing, settings in the former override those in the latter; in the event of neither file existing, PyXPlot uses its own default settings.

The configuration file should take the form of a series of sections, each headed by a section heading enclosed in square brackets, and followed by variables declared using the format:

```
OUTPUT=foo.eps
```

The following sections are used, although they do not all need to be present in any given file:

- `settings` – contains parameters similar to those which can be set with the `set` command. A complete list is given in section 4.4 below.
- `terminal` – contains parameters for altering the behaviour and appearance of PyXPlot's interactive terminal. A complete list is given in section 4.5.
- `variables` – contains variable definitions. Any variables defined in this section will be predefined in the PyXPlot mathematical environment upon startup.
- `functions` – contains function definitions.
- `colours` – contains a variable '`palette`', which should be set to a comma-separated list of the sequence of colours in the palette used to plot datasets. The first will be called colour 1 in PyXPlot, the second colour 2, etc. A list of recognised colour names is given in section 4.6.
- `latex` – contains a variable '`preamble`', which is prefixed to the beginning of all LaTeX text items, before the `\begin{document}` statement. It can be used to define custom LaTeX macros, or to include packages using the `\includepackage{}` command.

4.3 An Example Configuration File

As an example, the following is a configuration file which would represent PyXPlot's default configuration:

```
[settings]
ASPECT=1.0
AUTOASPECT=ON
AXESCOLOUR=Black
BACKUP=OFF
BAR=1.0
BOXFROM=0
BOXWIDTH=0
COLOUR=ON
DATASTYLE=points
DISPLAY=ON
DPI=300
ENHANCED=ON
FONTSIZE=0
FUNCSTYLE=lines
GRID=OFF
GRIDAXISX=1
```

```
GRIDAXISY=1
GRIDMAJCOLOUR=Grey60
GRIDMINCOLOUR=Grey90
KEY=ON
KEYCOLUMNS=1
KEYPOS=TOP RIGHT
KEY_XOFF=0.0
KEY_YOFF=0.0
LANDSCAPE=OFF
LINEWIDTH=1.0
MULTIPLOT=OFF
ORIGINX=0.0
ORIGINY=0.0
OUTPUT=
POINTLINEWIDTH=1.0
POINTSIZE=1.0
SAMPLES=250
TERMINVERT=OFF
TERMTRANSPARENT=OFF
TERMTYPE=X11_singlewindow
TEXTCOLOUR=Black
TEXTHALIGN=Left
TEXTVALIGN=Bottom
TITLE=
TIT_XOFF=0.0
TIT_YOFF=0.0
WIDTH=8.0

[terminal]
COLOUR=OFF
COLOUR_ERR=Red
COLOUR_REP=Green
COLOUR_WRN=Brown
SPLASH=ON

[variables]
pi = 3.14159265358979

[colours]
palette = Black, Red, Blue, Magenta, Cyan, Brown, Salmon, Gray,
Green, NavyBlue, Periwinkle, PineGreen, SeaGreen, GreenYellow,
Orange, CarnationPink, Plum

[latex]
```

PREAMBLE=

4.4 Configuration Options: `settings` section

The following table provides a brief description of the function of each of the parameters in the `settings` section of the above configuration file, with a list of possible values for each:

`ASPECT`	**Possible values:** Any floating-point number. **Analogous set command:** `set size ratio` Sets the aspect ratio of plots.
`AUTOASPECT`	**Possible values:** ON / OFF **Analogous set command:** `set size ratio` Sets whether plots have the automatic aspect ratio, which is the golden ratio. If `ON`, then the above setting is ignored.
`AXESCOLOUR`	**Possible values:** Any recognised colour. **Analogous set command:** `set axescolour` Sets the colour of axis lines and ticks.
`BACKUP`	**Possible values:** ON / OFF **Analogous set command:** `set backup` When this switch is set to 'ON', and plot output is being directed to file, attempts to write output over existing files cause a copy of the existing file to be preserved, with a tilda after its old filename (see section 3.3).
`BAR`	**Possible values:** Any floating-point number. **Analogous set command:** `set bar` Sets the horizontal length of the lines drawn at the end of errorbars, in units of their default length.
`BOXFROM`	**Possible values:** Any floating-point number. **Analogous set command:** `set boxfrom` Sets the horizontal point from which bars on bar charts appear to emanate.
`BOXWIDTH`	**Possible values:** Any floating-point number. **Analogous set command:** `set boxwidth` Sets the default width of boxes on barcharts. If negative, then the boxes have automatically selected widths, so that the interfaces between bars occur at the horizontal midpoints between the specified datapoints.

`COLOUR`	**Possible values:** ON / OFF **Analogous set command:** `set terminal` Sets whether output should be colour (ON) or monochrome (OFF).
`DATASTYLE`	**Possible values:** Any plot style. **Analogous set command:** `set data style` Sets the plot style used by default when plotting datafiles.
`DISPLAY`	**Possible values:** ON / OFF **Analogous set command:** `set display` When set to 'ON', no output is produced until the `set display` command is issued. This is useful for speeding up scripts which produce large multiplots; see section 3.5.1 for more details.
`DPI`	**Possible values:** Any floating-point number. **Analogous set command:** `set dpi` Sets the sampling quality used, in dots per inch, when output is sent to a bitmapped terminal (the jpeg/gif/png terminals).
`ENHANCED`	**Possible values:** ON / OFF **Analogous set command:** `set terminal` Sets whether the postscript terminal produces encapsulated postscript (ON), or printable postscript (OFF).
`FONTSIZE`	**Possible values:** Integers in the range $-4 \rightarrow 5$. **Analogous set command:** `set fontsize` Sets the fontsize of text, varying between LaTeX's `tiny` (-4) and `Huge` (5).
`FUNCSTYLE`	**Possible values:** Any plot style. **Analogous set command:** `set function style` Sets the plot style used by default when plotting functions.
`GRID`	**Possible values:** ON / OFF **Analogous set command:** `set grid` Sets whether a grid should be displayed on plots.
`GRIDAXISX`	**Possible values:** Any integer. **Analogous set command:** None Sets the default x-axis to which gridlines should attach, if the `set grid` command is called without specifying which axes to use.

`GRIDAXISY`	**Possible values:** Any integer. **Analogous set command:** None Sets the default y-axis to which gridlines should attach, if the `set grid` command is called without specifying which axes to use.
`GRIDMAJCOLOUR`	**Possible values:** Any recognised colour. **Analogous set command:** `set gridmajcolour` Sets the colour of major grid lines.
`GRIDMINCOLOUR`	**Possible values:** Any recognised colour. **Analogous set command:** `set gridmincolour` Sets the colour of minor grid lines.
`KEY`	**Possible values:** ON / OFF **Analogous set command:** `set key` Sets whether a legend is displayed on plots.
`KEYCOLUMNS`	**Possible values:** Any integer > 0. **Analogous set command:** `set keycolumns` Sets the number of columns into which the legends of plots should be divided.
`KEYPOS`	**Possible values:** "TOP RIGHT", "TOP MIDDLE", "TOP LEFT", "MIDDLE RIGHT", "MIDDLE MIDDLE", "MIDDLE LEFT", "BOTTOM RIGHT", "BOTTOM MIDDLE", "BOTTOM LEFT", "BELOW", "OUTSIDE". **Analogous set command:** `set key` Sets where the legend should appear on plots.
`KEY_XOFF`	**Possible values:** Any floating-point number. **Analogous set command:** `set key` Sets the horizontal offset, in approximate graph-widths, that should be applied to the legend, relative to its default position, as set by `KEYPOS`.
`KEY_YOFF`	**Possible values:** Any floating-point number. **Analogous set command:** `set key` Sets the vertical offset, in approximate graph-heights, that should be applied to the legend, relative to its default position, as set by `KEYPOS`.
`LANDSCAPE`	**Possible values:** ON / OFF **Analogous set command:** `set terminal` Sets whether output is in portrait orientation (OFF), or landscape orientation (ON).
`LINEWIDTH`	**Possible values:** Any floating-point number. **Analogous set command:** `set linewidth` Sets the width of lines on plots, as a multiple of the default.

`MULTIPLOT`	**Possible values:** ON / OFF **Analogous set command:** `set multiplot` Sets whether multiplot mode is on or off.
`ORIGINX`	**Possible values:** Any floating point number. **Analogous set command:** `set origin` Sets the horizontal position, in centimetres, of the default origin of plots on the page. Most useful when multiplotting many plots.
`ORIGINY`	**Possible values:** Any floating point number. **Analogous set command:** `set origin` Sets the vertical position, in centimetres, of the default origin of plots on the page. Most useful when multiplotting many plots.
`OUTPUT`	**Possible values:** Any string. **Analogous set command:** `set output` Sets the output filename for plots. If blank, the default filename of pyxplot.foo is used, where 'foo' is an extension appropriate for the file format.
`POINTLINEWIDTH`	**Possible values:** Any floating-point number. **Analogous set command:** `set pointlinewidth` / `plot with pointlinewidth` Sets the linewidth used to stroke points onto plots, as a multiple of the default.
`POINTSIZE`	**Possible values:** Any floating-point number. **Analogous set command:** `set pointsize` / `plot with pointsize` Sets the sizes of points on plots, as a multiple of their normal sizes.
`SAMPLES`	**Possible values:** Any integer. **Analogous set command:** `set samples` Sets the number of samples (datapoints) to be evaluated along the x-axis when plotting a function.
`TERMINVERT`	**Possible values:** ON / OFF **Analogous set command:** `set terminal` Sets whether jpeg/gif/png output has normal colours (OFF), or inverted colours (ON).
`TERMTRANSPARENT`	**Possible values:** ON / OFF **Analogous set command:** `set terminal` Sets whether jpeg/gif/png output has transparent background (ON), or solid background (OFF).

TERMTYPE	**Possible values:** X11_singlewindow, X11_multiwindow, PS, PNG, JPG, GIF **Analogous set command:** set terminal Sets whether output is sent to the screen or to disk, and, in the latter case, the format of the output. The ps option should be used for both encapsulated and normal postscript output; these are distinguished using the ENHANCED option, above.
TEXTCOLOUR	**Possible values:** Any recognised colour. **Analogous set command:** set textcolour Sets the colour of all text output.
TEXTHALIGN	**Possible values:** Left, Centre, Right **Analogous set command:** set texthalign Sets the horizontal alignment of text labels to their given reference positions.
TEXTVALIGN	**Possible values:** Top, Centre, Bottom **Analogous set command:** set textvalign Sets the vertical alignment of text labels to their given reference positions.
TITLE	**Possible values:** Any string. **Analogous set command:** set title Sets the title to appear at the top of the plot.
TIT_XOFF	**Possible values:** Any floating point number. **Analogous set command:** set title Sets the horizontal offset of the title of the plot from its default central location.
TIT_YOFF	**Possible values:** Any floating point number. **Analogous set command:** set title Sets the vertical offset of the title of the plot from its default location at the top of the plot.
WIDTH	**Possible values:** Any floating-point number. **Analogous set command:** set width / set size Sets the width of plots in centimetres.

4.5 Configuration Options: terminal section

The following table provides a brief description of the function of each of the parameters in the terminal section of the above configuration file, with a list of possible values for each:

`COLOUR`	**Possible values:** ON / OFF **Analogous commandline switches:** `-c`, `--colour`, `-m`, `--monochrome` Sets whether colour highlighting should be used in the interactive terminal. If turned on, output is displayed in green, warning messages in amber, and error messages in red; these colours are configurable, as described below. Note that not all UNIX terminals support the use of colour.
`COLOUR_ERR`	**Possible values:** Any recognised terminal colour. **Analogous commandline switches:** None. Sets the colour in which error messages are displayed when colour highlighting is used. Note that the list of recognised colour names differs from that used in PyXPlot; a list is given at the end of this section.
`COLOUR_REP`	**Possible values:** Any recognised terminal colour. **Analogous commandline switches:** None. As above, but sets the colour in which PyXPlot displays its non-error-related output.
`COLOUR_WRN`	**Possible values:** Any recognised terminal colour. **Analogous commandline switches:** None. As above, but sets the colour in which PyXPlot displays its warning messages.
`SPLASH`	**Possible values:** ON / OFF **Analogous commandline switches:** `-q`, `--quiet`, `-V`, `--verbose` Sets whether the standard welcome message is displayed upon startup.

The colours recognised by the `COLOUR_XXX` configuration options above are: `Red`, `Green`, `Brown`, `Blue`, `Purple`, `Magenta`, `Cyan`, `White`, `Normal`. The final option produces the default foreground colour of your terminal.

4.6 Recognised Colour Names

The following is a complete list of the colour names which PyXPlot recognises in the `set textcolour`, `set axescolour` commands, and in the `colours` section of its configuration file. It should be noted that they are case-insensitive:

GreenYellow, Yellow, Goldenrod, Dandelion, Apricot, Peach, Melon, YellowOrange, Orange, BurntOrange, Bittersweet, RedOrange, Mahogany, Maroon, BrickRed, Red, OrangeRed, RubineRed, WildStrawberry, Salmon, CarnationPink, Magenta, VioletRed, Rhodamine, Mulberry, RedViolet, Fuchsia, Lavender, Thistle, Orchid, DarkOrchid, Purple, Plum, Violet, RoyalPurple, BlueViolet, Periwinkle, CadetBlue, CornflowerBlue, MidnightBlue, NavyBlue, RoyalBlue, Blue, Cerulean, Cyan, ProcessBlue, SkyBlue, Turquoise, TealBlue, Aquamarine, BlueGreen, Emerald, JungleGreen, SeaGreen, Green, ForestGreen, PineGreen, LimeGreen, YellowGreen, SpringGreen, OliveGreen, RawSienna, Sepia, Brown, Tan, Gray, Grey, Black, White, white, black.

The following further colours provide a scale of shades of grey from dark to light, also case-insensitive:

grey05, grey10, grey15, grey20, grey25, grey30, grey35, grey40, grey45, grey50, grey55, grey60, grey65, grey70, grey75, grey80, grey85, grey90, grey95.

The US mis-spelling of grey (“gray”) is also accepted.

For a colour chart of these colours, the reader is referred to Appendix B of the *PyX Reference Manual*.[1]

[1] `http://pyx.sourceforge.net/manual/colorname.html`

Chapter 5

Examples

This chapter contains a few example PyXPlot plot scripts to illustrate its features. For each example, the plotting script is given, and an illustration of the resulting output.

5.1 Example 1: Plotting Functions – A Simple First Plot

As a simple first example, we plot two trigonometric functions. The syntax here is exactly as would have been used in the original gnuplot. The output is shown in figure 5.1.

PyXPlot Script:

```
# A very simple first example... plots sin(x)
# and cos(x)
reset
set xlabel 'x'
set ylabel 'y'
set term eps
set output 'examples/eps/example1.eps'
plot sin(x), cos(x)

# Produce a gif copy
set term gif
set dpi 207
set output 'examples/eps/example1.gif'
refresh
```

5.2 Example 2: Stacking Many Plots Together – Multiplot

In this example, we use the multiplot environment to produce a gallery of several plots. The `set origin` command is used to position each one. We also make use of multiple y-axes in the top-left plot: the functions $\sin(x)$ and $\sin^2(x)$ are plotting together, but on different y scales. The output is shown in figure 5.2.

PyXPlot Script:

```
# Example 2
# Uses multiplot to produce a gallery of
# trigonometric functions.

reset

set term eps
set output 'examples/eps/example2.eps'
set multiplot
set nodisplay
set xlabel 'x'
set ylabel 'y'

# Plot 0 (bottom left)
plot sin(x)

# Plot 1 (bottom right)
set origin 11,0
plot cos(x)

# Plot 2 (top left)
set origin 0,6.2
plot sin(x) ax x1y1, sin(x)**2 ax x1y2

# Plot 3 (top right)
set origin 11,6.2
plot sin(x)+cos(x)

# Now that we are finished preparing multiplot,
# turn display on
set display
refresh
```

```
# Produce a gif copy
set term gif
set dpi 98
set output 'examples/eps/example2.gif'
refresh
```

5.3 Example 3: Plotting A Datafile – Using Multiple Axes

This is a more complicated example. First of all, we plot two datafiles, one using a line, and another using points. We label our lines using arrows and text labels, using the same syntax that gnuplot uses. We also have multiple axes, this time having three x-axes on the same plot. The output is shown in figure 5.3.

PyXPlot Script:

```
# Example 3
# A rather more complicated plot to show off multiple axes,
# and LaTeXed labels on plots.
reset
# A few physical constants
min = 5
max = 200
phy_h = 6.626068e-34
phy_c = 3e8
phy_ev = 1.6e-19

# Set up plot basics...
set output 'examples/eps/example3.eps'
set terminal postscript eps monochrome
set grid
set key bottom right
set width 10
set log x
set log y
set title 'Simulated infrared dust spectrum for an \
{\mbox{\normalsize H\thinspace\footnotesize II}\kern3pt} region'

# X-axis is wavelength, lambda
set xlabel '$\lambda$/$\mu$m'

# Y-axis is emitted flux, integrated over grainsize a
set ylabel '$\int F_{\nu}(a)\mathrm{d}a \cdot 4\pi r^2 / \
\mathrm{W} \, \mathrm{Hz}^{-1}\, \mathrm{m}^2 \, \
\mathrm{H}^{-1}$'

# Make a second X-axis, in units of frequency, nu
set x2range [phy_c/(min*1e-6):phy_c/(max*1e-6)]
```

```
set log x2
set x2label '$\nu$/Hz'

# And a third X-axis, in units of photon energy, in eV
set x3range [phy_h*phy_c/(min*1e-6)/phy_ev:phy_h*phy_c/ \
(max*1e-6)/phy_ev]
set log x3
set x3label 'Photon Energy / eV'

# Put an arrow and label on our plot, labelling one
# of the lines
set arrow 1 from 60, 2e-5 to 38, 1e-5
set label 1 "$F_\nu=\nu^{\beta}B_\nu(30\mathrm{K})$" \
at 62, 1.8e-5

# Make f(x) a 30K greybody
T=30.0
h=6.626e-34
k=1.38e-23
c=3e8
f(x)=((c/(x*1e-6))**(3+2))/(exp(h*c/(x*1e-6*k*T))-1.0)

# Finally, plot all of our data
plot [min:max][1e-7:1e-3] 'examples/example2a.dat' using 1:2 \
t 'Nikoli\v{c}-Ford Dust Code' with lines, \
'examples/example2b.dat' t 'IRAS Photometry' using \
($1):(($2)/3e8*((($1)*1e-6)**2)*1.375191e+13/3.668333e+17), \
f(x)/f(60)*1.375191e+13/(3e8/(60e-6**2)) t '$\beta=2$ Greybody'

# Produce a gif copy
set term gif
set dpi 168
set output 'examples/eps/example3.gif'
refresh
```

5.4 Example 4: Something Completely Different

In this example, we demonstrate something rather different that PyXPlot can do. There is a common problem of trying to incorporate LaTeXed equations into various multimedia/graphics packages: the postscript format which LaTeX produces is not supported by programs such as Microsoft Powerpoint. PyXPlot offers a very quick and simple solution to this problem.

First of all, we set our terminal to produce png output. To overlay our output onto a Powerpoint slide, we will want it to have a transparent background, and so we also use the "transparent" terminal option (see section 3.2 for a discussion of PyXPlot terminal options). Finally, if we're producing a Powerpoint presentation with light-coloured text on a dark background, we will want to invert the colours to have white text, and so use the "invert" terminal option.

We can now produce plots which can readily be imported into Powerpoint. To produce LaTeXed equations, we use the multiplot environment's `text` command (see section 3.5).

Finally, as such a figure would not be very easy to incorporate into this User Manual, we produce a normal eps version of our equation, illustrating how to use the `refresh` command to produce multiple copies of the same figure in different graphic formats.

The output is shown in figure 5.4.

PyXPlot Script:

```
# Example 4
# Demonstrates how an equation might be output as a gif
# for inclusion in a slideshow in Microsoft Powerpoint.

reset

# Set terminal to produce transparent gif output
set term gif trans invert
set dpi 450
set output 'examples/eps/example4.gif'
set multiplot

# Render the Planck blackbody formula in LaTeX
text '$B_\nu = \frac{8\pi h}{c^3} \
\frac{\nu^3}{\exp \left( h\nu / kT \right) -1 }$' 0,0
text 'This is an example equation:' 0 , 0.75
```

```
# Produce a second copy of this plot as an eps file
set output 'examples/eps/example4.eps'
set term eps
refresh
```

5.5 Example 5: Multiplot – Linked Axes

In this example, we illustrate how to link the axes of plots on a multiplot, so that they share a common scale, and also demonstrate how to set the colours of datasets using the `with colour` plot modifier. In the top-right panel, we also make use of the multiplot environment to add a plot inset. Finally, we render this plot using the `landscape` terminal setting, showing how to fit more plot onto our sheet of paper. The output is shown in figure 5.5.

Notice how the linked axes autoscale intelligently. The right two plots both require larger vertical ranges than those plots to their lefts, to whose vertical axes they are linked. But once they are linked, the plots autoscale together, to ensure that they all have sufficient range for their data.

PyXPlot Script:

```
# Example 5
# A gallery of trigonometric functions demonstrating
# the use of linked axes.

reset
set term landscape eps
set output 'examples/eps/example5.eps'
set multiplot
set nodisplay

set xlabel '$x$'
set ylabel '$y$'
set xrange [-10.9:10.9]

width  = 8
height = 5.75

# Plot 0 (bottom left)
set key bottom right
set origin 0*width, 0*height
plot sin(x) with colour 3

# Plot 1 (bottom right)
set key top right
set origin 1*width, 0*height
set ylabel 'linkaxis 0'
plot cos(x)-1 with colour seagreen
```

```
# Plot 2 (top left)
set key top right
set origin 0*width, 1*height
set xlabel 'linkaxis 0'
set ylabel '$y$'
plot cos(x) with colour 7

# Plot 3 (top right)
set key bottom right
set origin 1*width, 1*height
set xlabel 'linkaxis 1'
set ylabel 'linkaxis 2'
plot sin(x)**2 + 1 with colour green

# Plot 4 (inset plot)
set xlabel ''
set ylabel ''
set key top ycentre
set fontsize -3
set origin 1.1*width, 1.15*height
set width width/3
p [-5:5] x**2

# Now that we are finished preparing multiplot,
# turn display on
set display
refresh

# Produce a gif copy
set term gif
set dpi 100
set output 'examples/eps/example5.gif'
refresh
```

5.6 Example 6: Bar Charts and Steps

In this example, we illustrate the `boxes`, `impulses` and `steps` plot styles, described in section 3.6, which operate similarly to how they operate in gnuplot. Panels (a) and (b) illustrates the `impulses` plot style for a sine wave, using the `set boxfrom` command to define the point from which the lines originate. Panel (c) illustrates the `fsteps` plot style, (d) `steps`, (e) `histeps` and (f) `boxes`. The output is shown in figure 5.6.

PyXPlot Script:

```
# Example 6
# A gallery showing different styles of barcharts/steps.
reset
set multiplot
set nodisplay
set samples 25
width=7
gold_ratio = 1/((1+sqrt(5))/2)

set terminal eps
set output 'examples/eps/example6.eps'
set width width
set xrange [-10.9:10.9]
set yrange [-1.2:1.2]
set nokey

# Plot 0 (bottom left)
set xlabel 'x'
set ylabel 'y'
set label 1 '(a)' at -9,0.8
set label 2 'histeps' -3.7,0.8
plot 'ex*/ex*e6.dat' with histeps, 'ex*/ex*e6.dat' with points

# Plot 1 (bottom right)
set origin 1*width, 0*width*gold_ratio
set xlabel 'x'
set ylabel 'linkaxis 0'
set label 1 '(b)' at -9,0.8
set label 2 'boxes' -3.7,0.8
plot 'ex*/ex*e6.dat' with boxes, 'ex*/ex*e6.dat' with points

# Plot 2 (middle left)
```

```
set origin 0*width, 1*width*gold_ratio
set xlabel 'linkaxis 0'
set ylabel 'y'
set label 1 '(c)' at -9,0.8
set label 2 'fsteps' -3.7,0.8
plot 'ex*/ex*e6.dat' with fsteps, 'ex*/ex*e6.dat' with points

# Plot 3 (middle right)
set origin 1*width, 1*width*gold_ratio
set xlabel 'linkaxis 1'
set ylabel 'linkaxis 2'
set label 1 '(d)' at -9,0.8
set label 2 'steps' -3.7,0.8
plot 'ex*/ex*e6.dat' with steps, 'ex*/ex*e6.dat' with points

# Plot 4 (top left)
set origin 0*width, 2*width*gold_ratio
set xlabel 'linkaxis 0'
set ylabel 'y'
set label 1 '(e)' at -9,0.8
set label 2 'impulses' -3.7,0.8
plot 'ex*/ex*e6.dat' with impulses, 'ex*/ex*e6.dat' with points

# Plot 5 (top right)
set origin 1*width, 2*width*gold_ratio
set boxfrom -0.5
set xlabel 'linkaxis 1'
set ylabel 'linkaxis 4'
set label 1 '(f)' at -9,0.8
set label 2 'boxes' -3.7,0.8
plot 'ex*/ex*e6.dat' with boxes, 'ex*/ex*e6.dat' with points

# Now that we are finished preparing multiplot,
# turn display on
set display
refresh

# Produce a gif copy
set term gif
set dpi 129
set output 'examples/eps/example6.gif'
refresh
```

5.7 Example 7: Bar Charts – Box Widths

In this example, we demonstrate different ways of specifying the widths of bars on a bar chart. In panel (a), the widths are automatically determined from the data, changing bar midway between datapoints. In panel (b), the `wboxes` plot style is used, which reads the widths of the bars from a third column in the datafile. In panel (c), we demonstrate how the `set boxfrom` command can be applied to bar charts, as well as to impulses. And in panel (d) we illustrate how the `fillcolour` modifier can be used to produce coloured bars. The output is shown in figure 5.7.

PyXPlot Script:

```
# Example 7
# Continued gallery of different barchart styles

reset
set multiplot
set nodisplay
width=7
gold_ratio = 1/((1+sqrt(5))/2)

set terminal eps
set output 'examples/eps/example7.eps'
set width width
set xrange [0.1:10.4]
set yrange [0:1.1]
set nokey

# Plot 0 (bottom left)
set xlabel 'x'
set ylabel 'y'
set label 1 '(a)' 8.2,0.9
plot 'examples/example7.dat' with boxes

# Plot 1 (bottom right)
set origin 1*width, 0*width*gold_ratio
set xlabel 'x'
set ylabel 'linkaxis 0'
set label 1 '(b)' 8.2,0.9
plot 'examples/example7.dat' with wboxes

# Plot 2 (top left)
```

```
set origin 0*width, 1*width*gold_ratio
set xlabel 'linkaxis 0'
set ylabel 'y'
set boxwidth 0.4
set label 1 '(c)' 8.2,0.9
plot 'examples/example7.dat' with boxes fc 2

# Plot 3 (top right)
set origin 1*width, 1*width*gold_ratio
set xlabel 'linkaxis 1'
set ylabel 'linkaxis 2'
set boxwidth 0.0
set boxfrom 0.5
set samples 40
set label 1 '(d)' 8.2,0.9
plot sin(x)*sin(x) with boxes fc 3 c 1, \
     cos(x)*cos(x) with boxes fc 2 c 1

# Now that we are finished preparing multiplot,
# turn display on
set display
refresh

# Produce a gif copy
set term gif
set dpi 131
set output 'examples/eps/example7.gif'
refresh
```

5.8 Example 8: Fitting Functions to Data

The `fit` command works in PyXPlot in essentially the same way as in gnuplot (see section 2.7). In this example, we take a series of data points, and first fit parabolas through them. For the first fit, $f(x)$, we do not take the errorbars into account; in the second, $g(x)$, we do. Then, we use the `spline` command to fit a spline, $h(x)$, through the same data (see section 3.8). Strong oscillation is seen in this example because of the angular nature of the data; it is not well-fit by a spline. The output is shown in figure 5.8.

PyXPlot Script:

```
# Example 8
# An example of fitting functions to a datafile.
reset

# Functional forms to be fitted -- parabolas
f(x) = a * x**2 + b * x + c
g(x) = d * x**2 + e * x + f

# First of all, fit data neglecting errorbars
fit f(x) 'examples/example8.dat' via a,b,c
# Now fit data taking errorbars into account
fit g(x) 'examples/example8.dat' using 1:2:3 via d,e,f
# Now fit a spline through the data
spline h() 'examples/example8.dat'

# Plot the resulting functions
set width 12
set key top xcentre
set xlabel 'x'
set ylabel 'y'
set term eps
set output 'examples/eps/example8.eps'
plot [0:8][0:5] \
     'examples/example8.dat' with yerrorbars, f(x), g(x), h(x)

# Produce a gif copy
set term gif
set dpi 154
set output 'examples/eps/example8.gif'
refresh
```

5.9 Example 9: Simple Examples of Function Splicing

Here, we demonstrate simple use of function splicing (see section 3.7). In panel (a), we plot the function $\sin(x)$, but specify that we only want it to be drawn in the range $-2 < x < 7$. In panel (b), we show how to define a discontinuous function similar to a top-hat function, also demonstrating how to set movable boundaries between the spliced components of functions, in this case using the variable a for this purpose.

Panels (c) and (d) demonstrate a more complex example, involving the splicing of a two-dimensional function.

PyXPlot Script:

```
# Example 9
# Two Simple Examples of Function Splicing

reset

set multiplot
set nodisplay
width=9
gold_ratio = 1/((1+sqrt(5))/2)
set terminal eps
set output 'examples/eps/example9.eps'
set width width

# Plot 0 (bottom left)
f(x)[-2:7] = sin(x)

set xlabel 'x'
set ylabel 'y'
set xrange [-10.9:10.9]
set label 1 '(a)' -9,0.8
plot f(x)

# Plot 1 (bottom right)
g(x,a)      =  a/10
g(x,a)[:-a] = -a/10
g(x,a)[a:]  = -a/10

set ylabel 'linkaxis 0'
set label 1 '(b)' -9,0.8
```

```
set origin width,0
set key bottom xcentre
plot g(x,2), g(x,5), g(x,7)

# Plot 2 (top left)
h(x,y) = 1
h(x,y)[1:][1:] = x*y
h(x,y)[1:][:1] = x
h(x,y)[:1][1:] = y

set nokey
set xlabel 'linkaxis 0'
set ylabel 'y'
set yrange [0.1:25]
set label 1 '(c)' -9,22
set origin 0,width*gold_ratio
plot h(x,cos(x)+1) w l

# Plot 3 (top right)
set xlabel 'linkaxis 1'
set ylabel 'linkaxis 2'
set label 1 '(d)' -9,22
set origin width,width*gold_ratio
plot h(x,min(tan(x),10)) w l

# Now that we are finished preparing multiplot,
# turn display on
set display
refresh

# Produce a gif copy
set term gif
set dpi 103
set output 'examples/eps/example9.gif'
refresh
```

5.10 Example 10: Removal of Unwanted Axes

In this example, we use the magic axis labels `nolabels`, `nolabelsticks` and `invisible`, which were described in section 3.3.1. In the lower-left plot, we show how to create a graph without mirrored x- and y-axes on the top and right sides of the plot. In the lower-right panel, we produce a plot with only x-axes visible, using them to produce a gallery showing the appearance resulting from the use of each of these magic labels. The top-left plot shows a simple sketch-graph with completely unlabelled axes. We also draw arrows over the top of the axes in this example, to give them arrowheads. Finally, in the top-right panel, we show one artistic application of plotting functions with no axes visible at all, creating a simple logo. The output is shown in figure 5.10.

PyXPlot Script:

```
# Example 10
# Example of the Removal of Unwanted Axes
reset

set multiplot
set nodisplay
set width 8
set terminal eps
set output 'examples/eps/example10.eps'

# Plot 0 (bottom left)
set x2label 'invisible'
set y2label 'invisible'
set xlabel 'x'
set ylabel 'y'
plot [0:5] (sin(x) ** 2)

# Plot 1 (bottom right)
set ylabel  'invisible'
set xlabel  'A normal axis'
set x3label 'An axis with outward-pointing ticks'
set x3ticdir outward
set x5label 'nolabels: A \texttt{nolabels} axis'
set x7label 'nolabelstics: A \texttt{nolabelstics} axis'
set x9label 'invisible: An \texttt{invisible} axis'
set origin 9.5,5.5
plot
```

```
# Plot 2 (top left)
unset label
unset axis x3x5x7x9
xmin = -0.5 ; xmax = 1.5
ymin = -0.5 ; ymax = 1.0
set arrow 1 from xmin,ymin to xmax,ymin
set arrow 2 from xmin,ymin to xmin,ymax
set arrow 3 from 0.6,0 to 0.5,0.20
set label 1 'A sketch of a parabola' at 0, -0.2
set xlabel 'nolabelstics'
set ylabel 'nolabelstics'
set nokey
set origin 0,6.5
plot [xmin:xmax][ymin:ymax] x ** 2

# Plot 3 (top right)
unset arrow
unset label
set xlabel 'invisible'
set ylabel 'invisible'
logo_x = 9.5
logo_y = 6.5
set textcolour Grey80
text '\large $\frac{\hbar^2}{2m}\frac{\partial^2 \psi}{\partial \
       x^2} + V\psi = E\psi$' logo_x+2.1, logo_y+0.5
text '\large $d \sin \theta = n\lambda$' logo_x+0.5, logo_y+3.5

set textcolour Grey70
text '\Large $\nabla \cdot D = \rho_{\mathrm{free}}$' \
      logo_x+2.9, logo_y+4.6
text '\Large $\nabla \times E = - \frac{\partial \bf B} \
      {\partial t}$' logo_x+1.2, logo_y+4.0
text '\Large $\nabla \cdot B = 0$' logo_x+0.9, logo_y+1.2
text '\Large $\nabla \times H = J_{\mathrm{free}} - \
      \frac{\partial \bf D}{\partial t}$' logo_x+3.8,logo_y+1.8

set textcolour Grey55
text '\Large $ds^2=\left(1-\frac{2GM}{rc^2}\right) \
      dt^2$' logo_x+0.4, logo_y+2
text '\large $H(t)=\frac{\dot R}{R}$' logo_x+6.1,logo_y+3.1
text '$q(t) = - \frac{\ddot R R}{\dot R^2}$' logo_x+5.3, logo_y+3.9
text '\large $d_\mathrm{L} = \left( \frac{L}{4\pi F} \right) \
      ^\frac{1}{2}$' logo_x+3.7, logo_y+1.2
```

```
text '\Large $\ddot x^a + \Gamma^a_{\phantom{a}bc} \
      \dot x^b \dot x^c = 0$' logo_x+4.5, logo_y+2.5

set textcolour Black
set label 1 '\Huge \textbf{PyXPlot}' at -8.5 , 0.05
set arrow 1 from 0.0 , -0.590 to 2.75 , -0.590 \
            with nohead lines linetype 3 colour 1
set arrow 2 from 2.5 , -0.590 to 2.5 , -0.325 with twoway
set label 2 '\Large ${\bf \Delta \phi}$' at 2.7, -0.5
set origin logo_x, logo_y
p [-9.5:4.8][-0.75:0.60] - x*exp(-x**2) + \
   (1/(exp((x-1)*3)+1) - 0.5)/4 - 0.2 with l lw 2 colour 1

# Now that we are finished preparing multiplot,
# turn display on
set display
refresh

# Produce a gif copy
set term gif
set dpi 103
set output 'examples/eps/example10.gif'
refresh
```

5.11 Example 11: The Arrows Plot Style

Here, we show two possible applications of the arrows plot style (see section 3.3). In the left panel, we plot a map of fluid flow around a vortex core, the dotted circle showing the outline of the vortex core. The source for this is a datafile mapping fluid velocity as a function of position. In the right panel, we show a series of datapoints before and after some correction factor is applied to them, showing how the data are moved in the process. The output of this example is shown in figure 5.11.

PyXPlot Script:

```
# Example 11
# Examples of the 'arrows' plotting style

reset

set multiplot
set nodisplay
width = 15
set terminal eps
set output 'examples/eps/example11.eps'
set width width
set size square
set fontsize 2
set nokey

# Plot 0 (left)
set xlabel 'x'
set ylabel 'y'
plot [-10.9:10.9][-10.9:10.9] \
 'examples/example11.dat' i 0 u 1:2:($1+$3):($2+$4) w arrows, \
 4*sin(x/10.9*pi):4*cos(x/10.9*pi) u 2:3 w lt 2 col black

# Plot 1 (right)
set origin width, 0
set ylabel 'linkaxis 0'
set key bottom right
plot [-10.9:10.9] \
 'examples/example11.dat' i 1 t '' with arrows, \
 'examples/example11.dat' i 1 t 'Before correction' u 1:2 w p, \
 'examples/example11.dat' i 1 t 'After  correction' u 3:4 w p
```

```
# Now that we are finished preparing multiplot,
# turn display on
set display
refresh

# Produce a gif copy
set term gif
set dpi 63
set output 'examples/eps/example11.gif'
refresh
```

5.12 Output Produced by Examples

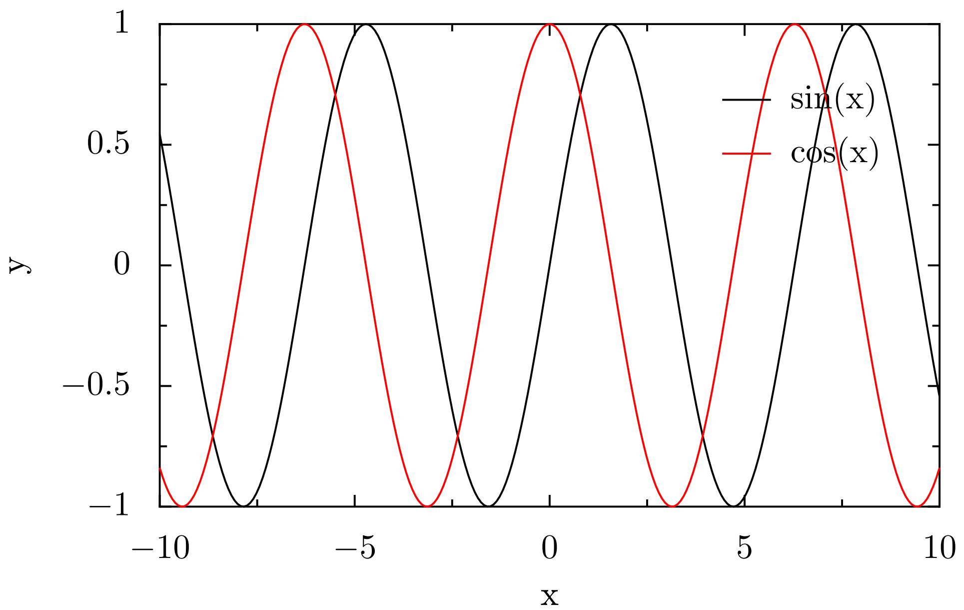

Figure 5.1: The output produced by example script 1, *Plotting Functions – A Simple First Plot.*

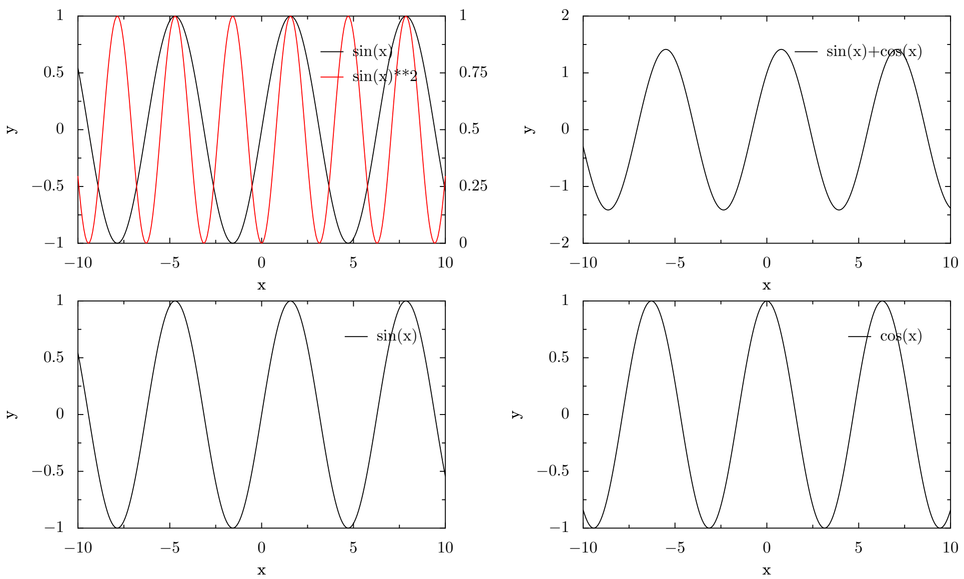

Figure 5.2: The output produced by example script 2, *Stacking Many Plots Together – Multiplot.*

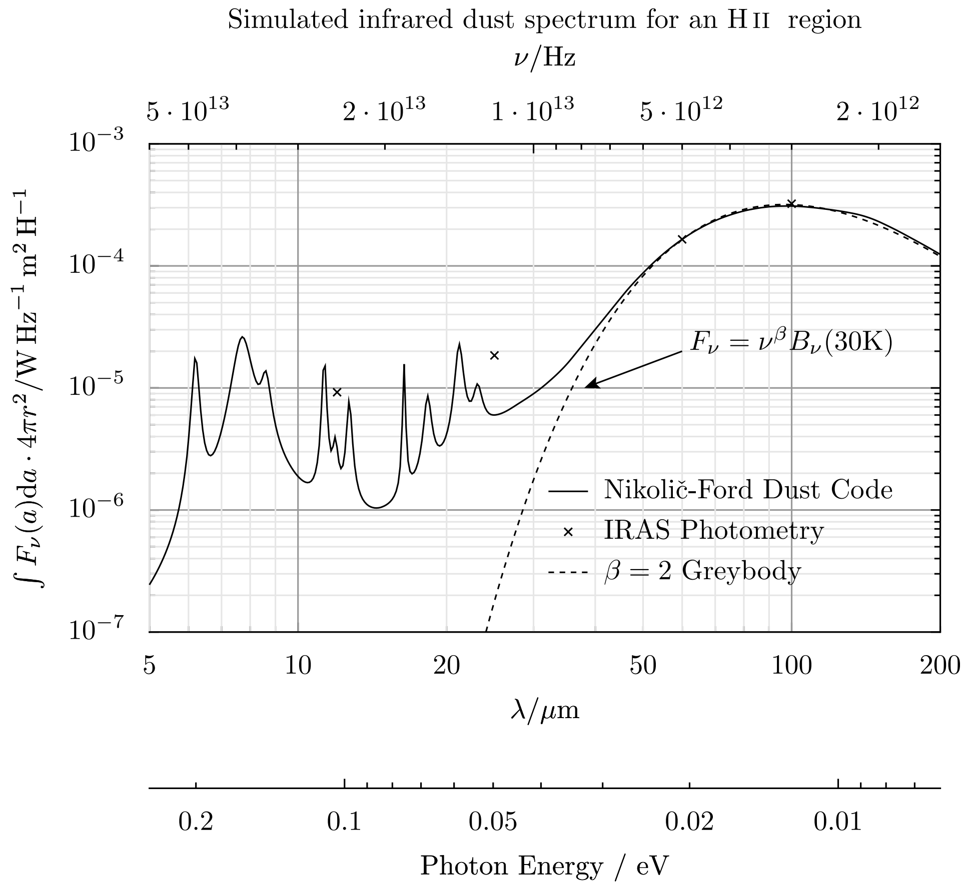

Figure 5.3: The output produced by example script 3, *Plotting A Datafile – Using Multiple Axes.*

This is an example equation:

$$B_\nu = \frac{8\pi h}{c^3}\frac{\nu^3}{\exp(h\nu/kT)-1}$$

Figure 5.4: The output produced by example script 4, *Something Completely Different.*

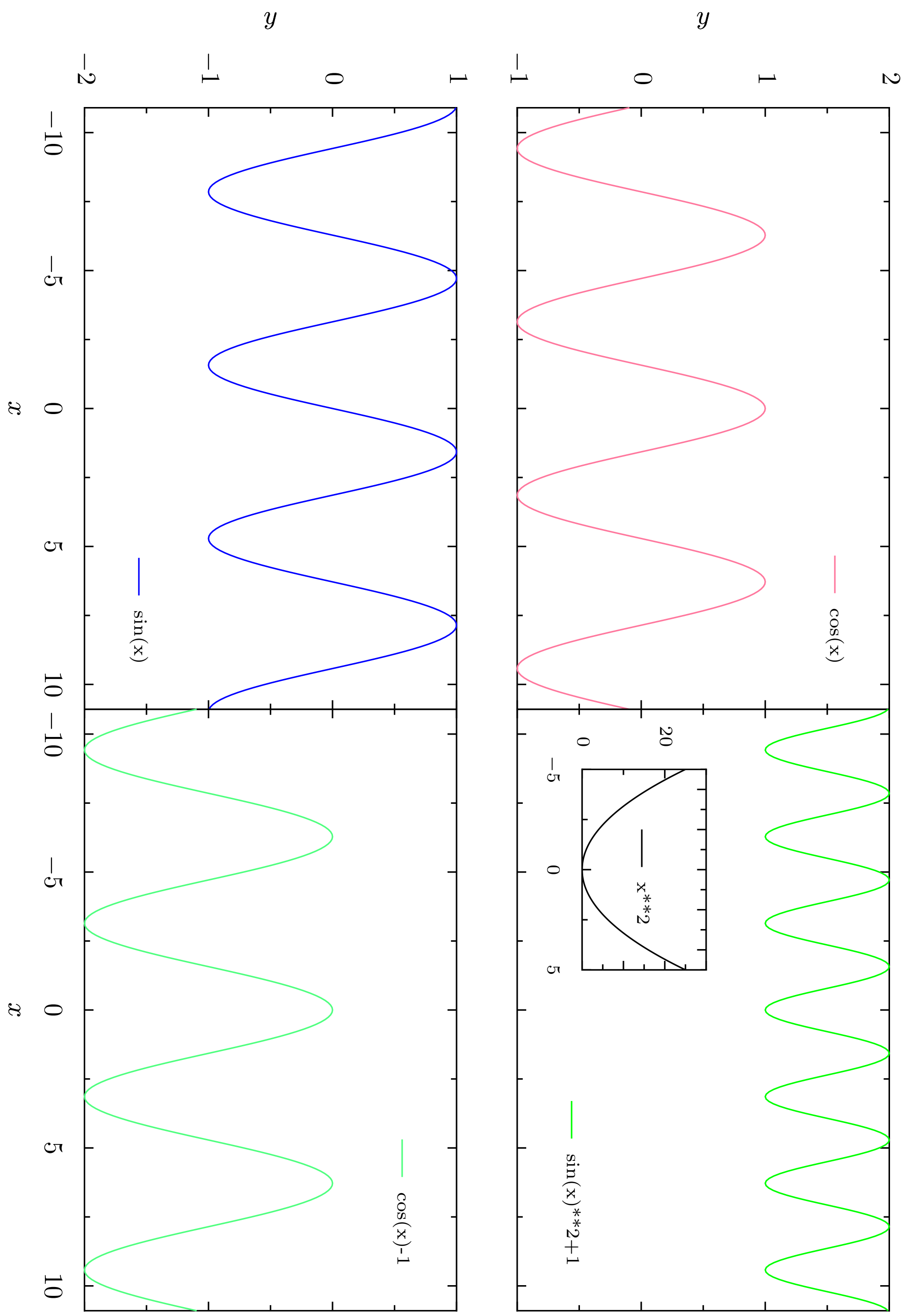

Figure 5.5: The output produced by example script 5, *Multiplot – Linked Axes.*

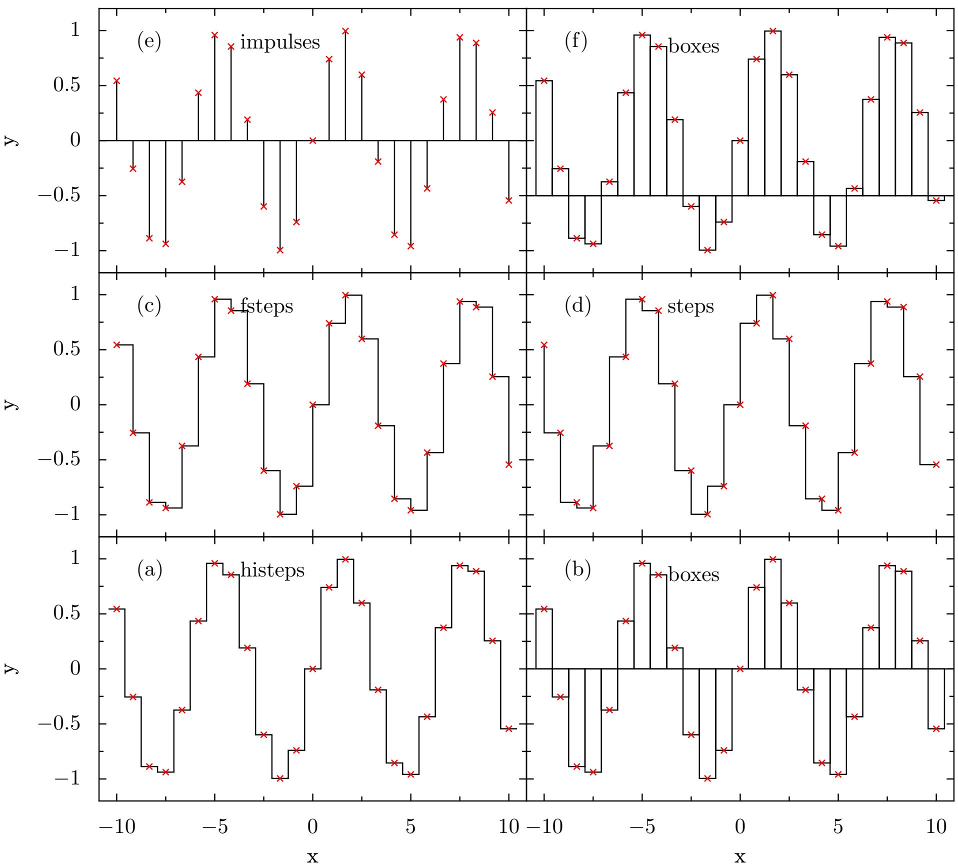

Figure 5.6: The output produced by example script 6, *Bar Charts and Steps*.

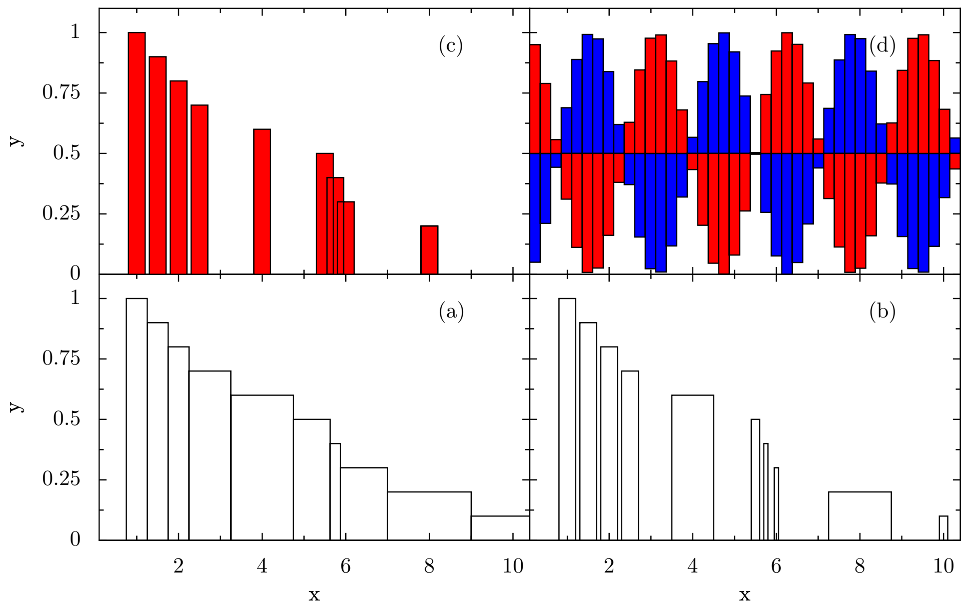

Figure 5.7: The output produced by example script 7, *Bar Charts – Box Widths*.

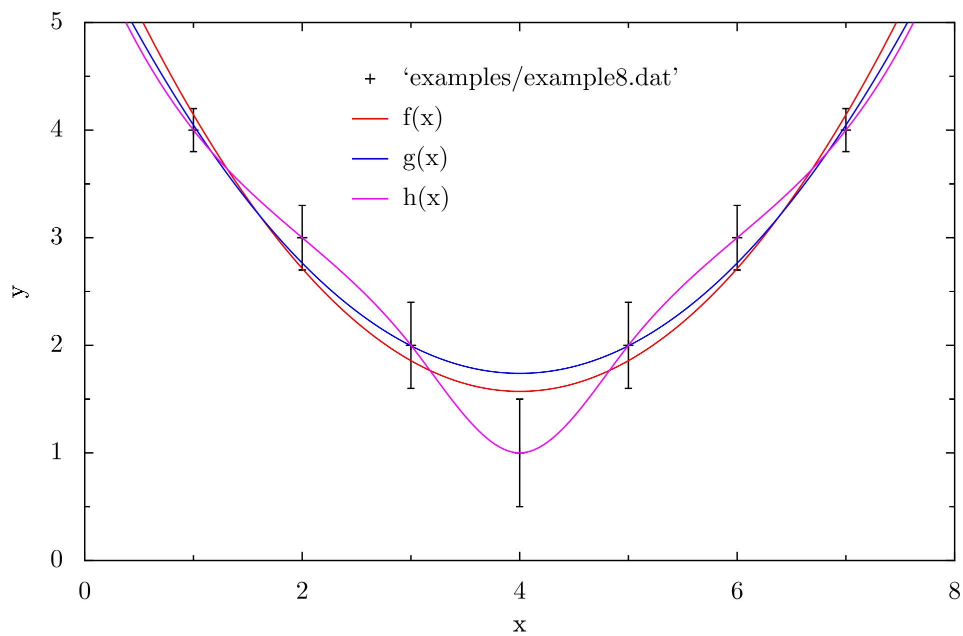

Figure 5.8: The output produced by example script 8, *Fitting Functions to Data*.

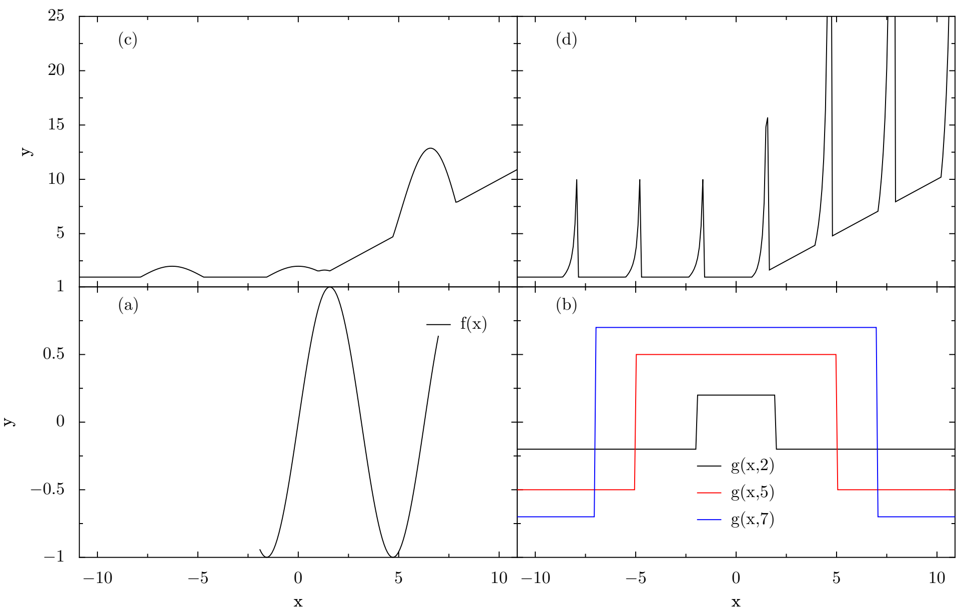

Figure 5.9: The output produced by example script 9, *Simple Examples of Function Splicing*.

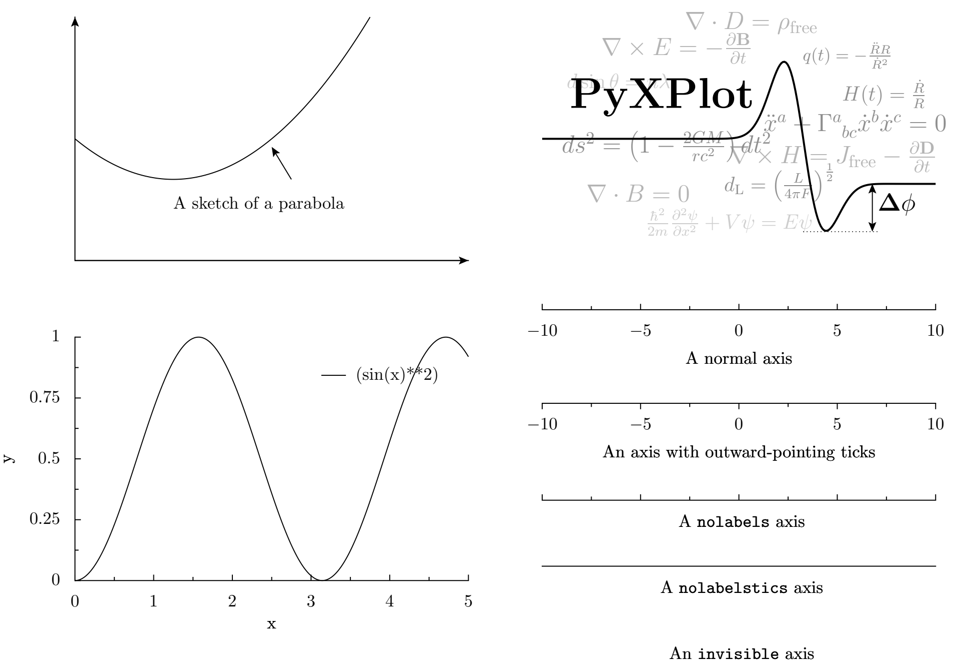

Figure 5.10: The output produced by example script 10, *Removal of Unwanted Axes*.

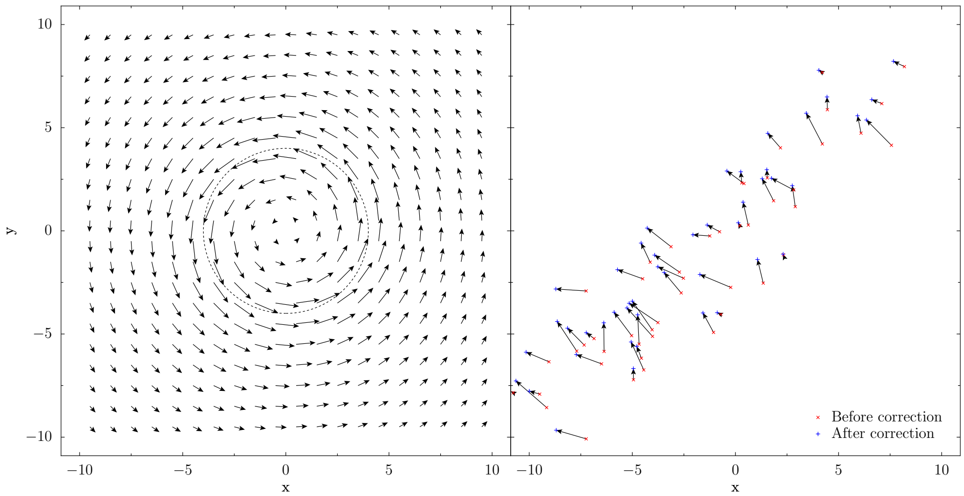

Figure 5.11: The output produced by example script 11, *The Arrows Plot Style*.

Chapter 6

The `fit` Command: Mathematical Details

In this section, the mathematical details of the workings of the `fit` command are described. This may be of interest in diagnosing its limitations, and also in understanding the various quantities that it outputs after a fit is found. This discussion must necessarily be a rather brief treatment of a large subject; for a fuller account, the reader is referred to D.S. Sivia's *Data Analysis: A Bayesian Tutorial*.

6.1 Notation

I shall assume that we have some function $f()$, which takes n_x parameters, $x_0 ... x_{n_\mathrm{x}-1}$, the set of which may collectively be written as the vector $\mathbf{x}$. We are supplied a datafile, containing a number n_d of datapoints, each consisting of a set of values for each of the n_x parameters, and one for the value which we are seeking to make $f(\mathbf{x})$ match. I shall call of parameter values for the ith datapoint $\mathbf{x}_i$, and the corresponding value which we are trying to match f_i. The datafile may contain error estimates for the values f_i, which I shall denote σ_i. If these are not supplied, then I shall consider these quantities to be unknown, and equal to some constant σ_data.

Finally, I assume that there are n_u coefficients within the function $f()$ that we are able to vary, corresponding to those variable names listed after the `via` statement in the `fit` command. I shall call these coefficients $u_0 ... u_{n_\mathrm{u}-1}$, and refer to them collectively as $\mathbf{u}$.

I model the values f_i in the supplied datafile as being noisy Gaussian-distributed observations of the true function $f()$, and within this framework, seek to find that vector of values $\mathbf{u}$ which is most probable, given these observations. The probability of any given $\mathbf{u}$ is written $\mathrm{P}\left(\mathbf{u} | \{\mathbf{x}_i, f_i, \sigma_i\}\right)$.

6.2 The Probability Density Function

Bayes' Theorem states that:

$$\mathrm{P}\left(\mathbf{u} \middle| \{\mathbf{x}_i, f_i, \sigma_i\}\right) = \frac{\mathrm{P}\left(\{f_i\} \middle| \mathbf{u}, \{\mathbf{x}_i, \sigma_i\}\right) \mathrm{P}\left(\mathbf{u} \middle| \{\mathbf{x}_i, \sigma_i\}\right)}{\mathrm{P}\left(\{f_i\} \middle| \{\mathbf{x}_i, \sigma_i\}\right)} \tag{6.1}$$

Since we are only seeking to maximise the quantity on the left, and the denominator, termed the Bayesian *evidence*, is independent of $\mathbf{u}$, we can neglect it and replace the equality sign with a proportionality sign. Furthermore, if we assume a uniform prior, that is, we assume that we have no prior knowledge to bias us towards certain more favoured values of $\mathbf{u}$, then $\mathrm{P}(\mathbf{u})$ is also a constant which can be neglected. We conclude that maximising $\mathrm{P}\left(\mathbf{u} \middle| \{\mathbf{x}_i, f_i, \sigma_i\}\right)$ is equivalent to maximising $\mathrm{P}\left(\{f_i\} \middle| \mathbf{u}, \{\mathbf{x}_i, \sigma_i\}\right)$.

Since we are assuming f_i to be Gaussian-distributed observations of the true function $f()$, this latter probability can be written as a product of n_d Gaussian distributions:

$$\mathrm{P}\left(\{f_i\} \middle| \mathbf{u}, \{\mathbf{x}_i, \sigma_i\}\right) = \prod_{i=0}^{n_\mathrm{d}-1} \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left(\frac{-\left[f_i - f_\mathbf{u}(\mathbf{x}_i)\right]^2}{2\sigma_i^2}\right) \tag{6.2}$$

The product in this equation can be converted into a more computationally workable sum by taking the logarithm of both sides. Since logarithms are monotonically increasing functions, maximising a probability is equivalent to maximising its logarithm. We may write the logarithm L of $\mathrm{P}\left(\mathbf{u} \middle| \{\mathbf{x}_i, f_i, \sigma_i\}\right)$ as:

$$L = \sum_{i=0}^{n_\mathrm{d}-1} \left(\frac{-\left[f_i - f_\mathbf{u}(\mathbf{x}_i)\right]^2}{2\sigma_i^2}\right) + k \tag{6.3}$$

where k is some constant which does not affect the maximisation process. It is this quantity, the familiar sum-of-square-residuals, that we numerically maximise to find our best-fitting set of parameters, which I shall refer to from here on as $\mathbf{u}^0$.

6.3 Estimating the Error in $\mathbf{u}^0$

To estimate the error in the best-fitting parameter values that we find, we assume $\mathrm{P}\left(\mathbf{u} \middle| \{\mathbf{x}_i, f_i, \sigma_i\}\right)$ to be approximated by an n_u-dimensional Gaussian distribution around $\mathbf{u}^0$. Taking a Taylor expansion of $L(\mathbf{u})$ about $\mathbf{u}^0$, we can write:

$$
\begin{aligned}
L(\mathbf{u}) \quad = \quad & L(\mathbf{u}^0) + \underbrace{\sum_{i=0}^{n_\mathrm{u}-1} \left(u_i - u_i^0\right) \left.\frac{\partial L}{\partial u_i}\right|_{\mathbf{u}^0}}_{\text{Zero at } \mathbf{u}^0 \text{ by definition}} + \\
& \sum_{i=0}^{n_\mathrm{u}-1} \sum_{j=0}^{n_\mathrm{u}-1} \frac{\left(u_i - u_i^0\right)\left(u_j - u_j^0\right)}{2} \left.\frac{\partial^2 L}{\partial u_i \partial u_j}\right|_{\mathbf{u}^0} + \mathcal{O}\left(\mathbf{u} - \mathbf{u}^0\right)^3
\end{aligned}
\tag{6.4}
$$

Since the logarithm of a Gaussian distribution is a parabola, the quadratic terms in the above expansion encode the Gaussian component of the probability distribution $\mathrm{P}\left(\mathbf{u}|\left\{\mathbf{x}_i, f_i, \sigma_i\right\}\right)$ about $\mathbf{u}^0$.[1] We may write the sum of these terms, which we denote Q, in matrix form:

$$
Q = \frac{1}{2}\left(\mathbf{u} - \mathbf{u}^0\right)^{\mathbf{T}} \mathbf{A} \left(\mathbf{u} - \mathbf{u}^0\right) \tag{6.5}
$$

where the superscript $^{\mathbf{T}}$ represents the transpose of the vector displacement from $\mathbf{u}^0$, and $\mathbf{A}$ is the Hessian matrix of L, given by:

$$
A_{ij} = \nabla\nabla L = \left.\frac{\partial^2 L}{\partial u_i \partial u_j}\right|_{\mathbf{u}^0} \tag{6.6}
$$

This is the Hessian matrix which is output by the `fit` command. In general, an n_u-dimensional Gaussian distribution such as that given by equation (6.4) yields elliptical contours of equiprobability in parameter space, whose principal axes need not be aligned with our chosen coordinate axes – the variables $u_0 ... u_{n_u - 1}$. The eigenvectors $\mathbf{e}_i$ of $\mathbf{A}$ are the principal axes of these ellipses, and the corresponding eigenvalues λ_i equal $1/\sigma_i^2$, where σ_i is the standard deviation of the probability density function along the direction of these axes.

This can be visualised by imagining that we diagonalise $\mathbf{A}$, and expand equation (6.5) in our diagonal basis. The resulting expression for L is a sum of square terms; the cross terms vanish in this basis by definition. The equations of the equiprobability contours become the equations of ellipses:

$$
Q = \frac{1}{2} \sum_{i=0}^{n_\mathrm{u}-1} A_{ii} \left(u_i - u_i^0\right)^2 = k \tag{6.7}
$$

where k is some constant. By comparison with the equation for the logarithm of a Gaussian distribution, we can associate A_{ii} with $-1/\sigma_i^2$ in our eigenvector basis.

[1] The use of this is called *Gauss' Method*. Higher order terms in the expansion represent any non-Gaussianity in the probability distribution, which we neglect. See MacKay, D.J.C., *Information Theory, Inference and Learning Algorithms*, CUP (2003).

The problem of evaluating the standard deviations of our variables u_i is more complicated, however, as we are attempting to evaluate the width of these elliptical equiprobability contours in directions which are, in general, not aligned with their principal axes. To achieve this, we first convert our Hessian matrix into a covariance matrix.

6.4 The Covariance Matrix

The terms of the covariance matrix V_{ij} are defined by:

$$V_{ij} = \left\langle \left(u_i - u_i^0\right) \left(u_j - u_j^0\right) \right\rangle \tag{6.8}$$

Its leading diagonal terms may be recognised as equalling the variances of each of our n_u variables; its cross terms measure the correlation between the variables. If a component $V_{ij} > 0$, it implies that higher estimates of the coefficient u_i make higher estimates of u_j more favourable also; if $V_{ij} < 0$, the converse is true.

It is a standard statistical result that $\mathbf{V} = (-\mathbf{A})^{-1}$. In the remainder of this section we prove this; readers who are willing to accept this may skip onto section 6.5.

Using Δu_i to denote $\left(u_i - u_i^0\right)$, we may proceed by rewriting equation (6.8) as:

$$\begin{aligned} V_{ij} &= \int \cdots \int_{u_i=-\infty}^{\infty} \Delta u_i \Delta u_j \mathrm{P}\left(\mathbf{u} \middle| \{\mathbf{x}_i, f_i, \sigma_i\}\right) \mathrm{d}^{n_\mathrm{u}}\mathbf{u} \qquad (6.9) \\ &= \frac{\int \cdots \int_{u_i=-\infty}^{\infty} \Delta u_i \Delta u_j \exp(-Q)\, \mathrm{d}^{n_\mathrm{u}}\mathbf{u}}{\int \cdots \int_{u_i=-\infty}^{\infty} \exp(-Q)\, \mathrm{d}^{n_\mathrm{u}}\mathbf{u}} \end{aligned}$$

The normalisation factor in the denominator of this expression, which we denote as Z, the *partition function*, may be evaluated by n_u-dimensional Gaussian integration, and is a standard result:

$$\begin{aligned} Z &= \int \cdots \int_{u_i=-\infty}^{\infty} \exp\left(\frac{1}{2}\Delta\mathbf{u}^\mathbf{T}\mathbf{A}\Delta\mathbf{u}\right) \mathrm{d}^{n_\mathrm{u}}\mathbf{u} \qquad (6.10) \\ &= \frac{(2\pi)^{n_\mathrm{u}/2}}{\mathrm{Det}(-\mathbf{A})} \end{aligned}$$

Differentiating $\log_e(Z)$ with respect of any given component of the Hessian matrix A_{ij} yields:

$$-2\frac{\partial}{\partial A_{ij}}\left[\log_e(Z)\right] = \frac{1}{Z} \int \cdots \int_{u_i=-\infty}^{\infty} \Delta u_i \Delta u_j \exp(-Q)\, \mathrm{d}^{n_\mathrm{u}}\mathbf{u} \tag{6.11}$$

which we may identify as equalling V_{ij}:

$$\begin{aligned} V_{ij} &= -2\frac{\partial}{\partial A_{ij}}\left[\log_e(Z)\right] \\ &= -2\frac{\partial}{\partial A_{ij}}\left[\log_e((2\pi)^{n_\mathrm{u}/2}) - \log_e(\mathrm{Det}(-\mathbf{A}))\right] \\ &= 2\frac{\partial}{\partial A_{ij}}\left[\log_e(\mathrm{Det}(-\mathbf{A}))\right] \end{aligned} \tag{6.12}$$

This expression may be simplified by recalling that the determinant of a matrix is equal to the scalar product of any of its rows with its cofactors, yielding the result:

$$\frac{\partial}{\partial A_{ij}}\left[\mathrm{Det}(-\mathbf{A})\right] = -a_{ij} \tag{6.13}$$

where a_{ij} is the cofactor of A_{ij}. Substituting this into equation (6.12) yields:

$$V_{ij} = \frac{-a_{ij}}{\mathrm{Det}(-\mathbf{A})} \tag{6.14}$$

Recalling that the adjoint $\mathbf{A}^\dagger$ of the Hessian matrix is the matrix of cofactors of its transpose, and that $\mathbf{A}$ is symmetric, we may write:

$$V_{ij} = \frac{-\mathbf{A}^\dagger}{\mathrm{Det}(-\mathbf{A})} \equiv (-\mathbf{A})^{-1} \tag{6.15}$$

which proves the result stated earlier.

6.5 The Correlation Matrix

Having evaluated the covariance matrix, we may straightforwardly find the standard deviations in each of our variables, by taking the square roots of the terms along its leading diagonal. For datafiles where the user does not specify the standard deviations σ_i in each value f_i, the task is not quite complete, as the Hessian matrix depends critically upon these uncertainties, even if they are assumed the same for all of our f_i. This point is returned to in section 6.6.

The correlation matrix $\mathbf{C}$, whose terms are given by:

$$C_{ij} = \frac{V_{ij}}{\sigma_i \sigma_j} \tag{6.16}$$

may be considered a more user-friendly version of the covariance matrix for inspecting the correlation between parameters. The leading diagonal terms are all clearly equal unity by construction. The cross terms lie in the range $-1 \leq C_{ij} \leq 1$, the upper limit of this range representing perfect correlation between parameters, and the lower limit perfect anti-correlation.

6.6 Finding σ_i

Throughout the preceding sections, the uncertainties in the supplied target values f_i have been denoted σ_i (see section 6.1). The user has the option of supplying these in the source datafile, in which case the provisions of the previous sections are now complete; both best-estimate parameter values and their uncertainties can be calculated. The user may also, however, leave the uncertainties in f_i unstated, in which case, as described in section 6.1, we assume all of the data values to have a common uncertainty σ_{data}, which is an unknown.

In this case, where $\sigma_i = \sigma_{\mathrm{data}} \,\forall\, i$, the best fitting parameter values are independent of σ_{data}, but the same is not true of the uncertainties in these values, as the terms of the Hessian matrix do depend upon σ_{data}. We must therefore undertake a further calculation to find the most probable value of σ_{data}, given the data. This is achieved by maximising $\mathrm{P}\left(\sigma_{\mathrm{data}}\middle|\left\{\mathbf{x}_i, f_i\right\}\right)$. Returning once again to Bayes' Theorem, we can write:

$$\mathrm{P}\left(\sigma_{\mathrm{data}}\middle|\left\{\mathbf{x}_i, f_i\right\}\right) = \frac{\mathrm{P}\left(\left\{f_i\right\}\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}\right)\mathrm{P}\left(\sigma_{\mathrm{data}}\middle|\left\{\mathbf{x}_i\right\}\right)}{\mathrm{P}\left(\left\{f_i\right\}\middle|\left\{\mathbf{x}_i\right\}\right)} \tag{6.17}$$

As before, we neglect the denominator, which has no effect upon the maximisation problem, and assume a uniform prior $\mathrm{P}\left(\sigma_{\mathrm{data}}\middle|\left\{\mathbf{x}_i\right\}\right)$. This reduces the problem to the maximisation of $\mathrm{P}\left(\left\{f_i\right\}\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}\right)$, which we may write as a marginalised probability distribution over $\mathbf{u}$:

$$\begin{aligned}\mathrm{P}\left(\left\{f_i\right\}\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}\right) = \int \cdots \int_{-\infty}^{\infty} \; & \mathrm{P}\left(\left\{f_i\right\}\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}, \mathbf{u}\right) \times \\ & \mathrm{P}\left(\mathbf{u}\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}\right) \mathrm{d}^{n_{\mathrm{u}}}\mathbf{u}\end{aligned} \tag{6.18}$$

Assuming a uniform prior for $\mathbf{u}$, we may neglect the latter term in the integral, but even with this assumption, the integral is not generally tractable, as $\mathrm{P}\left(\left\{f_i\right\}\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}, \left\{\mathbf{u}_i\right\}\right)$ may well be multimodal in form. However, if we neglect such possibilities, and assume this probability distribution to be approximate a Gaussian *globally*, we can make use of the standard result for an n_{u}-dimensional Gaussian integral:

$$\int \cdots \int_{-\infty}^{\infty} \exp\left(\frac{1}{2}\mathbf{u}^{\mathbf{T}}\mathbf{A}\mathbf{u}\right) \mathrm{d}^{n_{\mathrm{u}}}\mathbf{u} = \frac{(2\pi)^{n_{\mathrm{u}}/2}}{\sqrt{\mathrm{Det}\left(-\mathbf{A}\right)}} \tag{6.19}$$

We may thus approximate equation (6.18) as:

$$\begin{aligned}\mathrm{P}\left(\left\{f_i\right\}\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}\right) \approx \; & \mathrm{P}\left(\left\{f_i\right\}\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}, \mathbf{u}^0\right) \times \\ & \mathrm{P}\left(\mathbf{u}^0\middle|\sigma_{\mathrm{data}}, \left\{\mathbf{x}_i, f_i\right\}\right) \frac{(2\pi)^{n_{\mathrm{u}}/2}}{\sqrt{\mathrm{Det}\left(-\mathbf{A}\right)}}\end{aligned} \tag{6.20}$$

As in section 6.2, it is numerically easier to maximise this quantity via its logarithm, which we denote L_2, and can write as:

$$L_2 = \sum_{i=0}^{n_\mathrm{d}-1}\left(\frac{-\left[f_i - f_{\mathbf{u}^0}(\mathbf{x}_i)\right]^2}{2\sigma_\mathrm{data}^2} - \log_e\left(2\pi\sqrt{\sigma_\mathrm{data}}\right)\right) + \log_e\left(\frac{(2\pi)^{n_\mathrm{u}/2}}{\sqrt{\mathrm{Det}\left(-\mathbf{A}\right)}}\right) \tag{6.21}$$

This quantity is maximised numerically, a process simplified by the fact that $\mathbf{u}^0$ is independent of σ_data.

Chapter 7

ChangeLog

2006 Sep 09: PyXPlot 0.5.8

- Many bugfixes to error trapping and reporting.

2006 Aug 26: PyXPlot 0.5.7

- set display command implemented.
- set keycolumns command implemented.
- CTRL-C behaviour changed; no longer quits PyXPlot.
- plot '*.dat' now arranges files alphabetically.
- Escaping of LaTeX $<$ and $>$ symbols fixed.
- Major bugfix to fit command's error estimation.
- Major bugfix to the positioning of legends in the "outside" and "below" positions to avoid overlapping with axes.
- help command text substantially revised.

2006 Aug 18: PyXPlot 0.5.6

- Ability to unset variables via "a=" implemented.
- Handling on scipy error messages in the int_dx and spline commands improved.
- Colour-highlighted terminal added.
- The inline help system made much more complete.
- select modifier implemented.
- set texthalign and set textvalign implemented.

- set xticdir command implemented.
- Support for CSV input datafiles implemented.
- pyxplot_watch quiet mode added. Also, behaviour changed to allow the watching of files, even when they do not initially exist.
- Labels can now be placed on "nolabels", "nolabelstics" and "invisible" axes. Example 10 changed to demonstrate this.
- set log, when issued on its own, now applies to all axes, rather than throwing an error.

2006 Jul 25: PyXPlot 0.5.5

- pyxplot_watch implemented.
- fit command now gives error estimates, as well as correlation matrices.
- Many new pointtypes added, including upper and lower limit symbols.
- Handling of SIGINT improved; now exits current command in interactive mode, and exits PyXPlot when running a script.
- Quote characters can now be escaped in LaTeX strings, to allow strings with both ' and " characters to be rendered.
- Installer no longer creates any files belonging to root in the user's homespace.
- show xlabel and show xrange implemented.
- Bug fix: cd command no longer crashes if target directory doesn't exist.
- Bug fix: some commands, e.g. plot, which previously didn't work when capitalised, now do.
- Major bug fix to int_dx and diff_dx functions.

2006 Jul 3: PyXPlot 0.5.4

- edit command implemented.
- Numerical integration and differentiation functions implemented.
- New makefile installer added.
- man page added.
- Brief tour of gnuplot syntax added to documentation.

- Many minor bug fixes.

2006 Jun 27: PyXPlot 0.5.3

- set bar and set palette implemented.
- Stacked barcharts implemented.
- Command history files and the save command implemented.
- Plotting of functions with errorbars implemented.
- Ability to define a LaTeX preamble implemented.
- Bug fix to smoothed splines, to ensure that smoothing is always applied to a sensible degree by default.
- Bug fix to the autoscaling of bar charts, histograms and errorbars, to ensure that their full extent is contained within the plot area.
- Bug fix to arrow plotting, to prevent PyX from crashing if arrows of zero lengths are plotting (they have no direction...)

2006 Jun 14: PyXPlot 0.5.2

- spline command, and csplines/acsplines plot styles implemented.
- Syntax plot[0:1], with no space, now allowed.
- Automatic names of datasets in legends no longer have full paths, but only the path in the form that the user specified it.
- Bug fix to the handling of LaTeX special characters in the automatic names of datasets, e.g. file paths containing underscores.
- Error messages now sent to stderr, rather than stdout.
- multiplot mode now plots items in the order that they are plotted; previously all arrows and text labels had been plotted in front of plots.
- set backup command implemented, for keeping backups of overwritten files.
- Bug fix, enabling the use of axis x5 without axis x3, and likewise for y.
- unset axis command implemented, for removing axes from plots.
- ‘invisible’, ‘nolabels’, and ‘nolabelsticks’ axis title implemented, for producing axes without text labels.

- plot 'every' modifier re-implemented, to use the same syntax as gnuplot.
- fit command re-implemented to work with functions of > 1 variable.
- plot with pointlines defined as alias for 'linespoints'.
- plot using rows syntax implemented, for plotting horizontally-arranged datafiles.
- Bug fix to replot command in multiplot mode, to take account of any move commands applied to the last plot.
- Bug fix to errorbar pointsizes. pointsize modifier now produces sensible output with all errorbar plot styles.
- show command re-implemented to accept any word that the set command will.

2006 Jun 2: PyXPlot 0.5.1

- Pling and cd commands implemented; ‘ ‘ shell command substitution implemented.
- Arrows (both from set arrow and the arrow command) can now have linetypes and colours set.
- Colours can now be specified as either palette indices or PyX colour names in all contexts – e.g. 'plot with colour red'.
- Function plotting fixed to allow plotting of functions which are not valid across the whole range of the x-axis.
- Transparent terminals now have anti-aliasing disabled.
- Warnings now issued when too many columns are specified in plot command; duplicate errors filtered out in two-pass plotting.
- Function splicing implemented.
- Documentation: sections on barcharts, function splicing, and datafile globbing added.

2006 May 27: PyXPlot 0.5.0

- Name changed to PyXPlot.
- Change to distribution format: PyX Version 0.9 now ships with package.
- Safety installer added; checks for required packages.

- ‘errorrange’ plot styles added; allow errorbars to be given as min/max values, rather than as a standard deviation.
- ‘boxes’, ‘wboxes’, ‘steps’, ‘fsteps’, ‘histeps’ and ‘impulses’ plot styles implemented – allow the production of histograms and bar charts.
- plot with fillcolour implemented, to allow coloured bar charts.
- Handling of broken datafiles sanitised: now warns for each broken line.
- gridlines on multiple axes, e.g. ‘set grid x1x2x3’ now allowed.
- Major bugfix to the way autoscaling works: linked axes share information and scale intelligently between plots.
- –help and –version commandline options implemented.
- ‘using’ specifiers for datafiles can now include expressions, such as $(2+x).
- eps terminal fixed to produce encapsulated postscript.
- datafile names now glob, so that plot ‘*’ will plot many datafiles.
- Documentation: examples 6,7 and 8 added.

2006 May 18: GnuPlot+ 0.4.3

- text and arrow commands now accept expressions rather than just floats for positional coordinates.
- clear command major bug-fixed.
- ‘plot with’ clause bugfixed; state variable was not resetting.
- Automatical key titles for datafile datasets made more informative.
- Autoscaling of multiple axes bugfixed.
- Autoscaling of inverted axes fixed.
- set grid command fixed to only produce x/y gridlines when requested.
- X11_singlewindow changed to use `gv --watch`.
- landscape terminal postscript header detection bugfixed.
- noenhanced terminal changed to produce proper postscript.
- Plotting of single column datafiles without using specifier fixed.

2006 May 4: GnuPlot+ 0.4.2

- Autoscaling redesigned, no longer uses PyX for this.
- Numerical expression handling fixed in set title, set origin and set label.
- Handling of children fixed, to prevent zombies from lingering around.
- arrow command implemented.
- set textcolour, set axescolour, set gridmajcolour, set gridmincolour and set fontsize implemented.
- Colour palette can now be set in configuration file.
- Ranges for axes other then x1/y1 can now be set in the plot command.
- Postscript noenhanced can now produce plots almost as big as an A4 sheet.
- Plotting of one column datafiles, against datapoint number, implemented.
- Negative errorbars error trapped.
- Comment lines now allowed in command files.

2006 May 1: GnuPlot+ 0.4.1

- Documentation converted from ASCII to LaTeX.
- ChangeLog added.
- Configuration files now supported.
- Prevention of temporary files in /tmp overwriting pre-existing files.
- set term enhanced / noenhanced / landscape / portrait / png / gif / jpeg / transparent / solid / invert / noinvert implemented.
- set dpi implemented, to allow user to choose quality of gif/jpg/png output.
- 'set grid' command now allows user to specify which axes grid attaches to (extended API).
- Support introduced for plotting gzipped datafiles. Filenames ending in '.gz' are assumed to be gzipped.
- load command implemented.
- move command implemented.

- Long lines can now be split using ‘\́linesplit character at the end of a line. Any whitespace at the beginning of the next is omitted.
- text / delete_text / undelete_text / move_text commands implemented.
- refresh command implemented. (extended API)
- point types, line styles, and colours now start at 1, for gnuplot compatibility.
- default terminal changed to postscript for non-interactive sessions.

2006 Apr 27: GnuPlot+ 0.4.0

- Bug fix: now looks for input scripts in the user’s cwd, not in /tmp.
- ‘set logscale’ is now valid syntax (as in gnuplot), as well as ‘set log’.
- multiplot implemented, including linked axes, though with some brokenness if linked axes are allowed to autoscale.
- ‘dots’ plotting style implemented.
- Bug fix: can now include a plot ‘with’ clause after an ‘axes’ clause; could not previously without an error message arising.
- Pointstyles now increment between plotted datasets, even in a colour terminal where the colours also increment.
- garbage collection of .eps files from the X11 terminal added. Previously they were left to fester in /tmp.
- pointlinewidth added as a plot style, specifying the linewidth to be used in plotting points. ‘set plw’ and ‘set lw’ both added (extended API).
- delete, clear and undelete commands added to the multiplot environment.
- unset command implemented.
- set notitle implemented.

2006 Apr 14: GnuPlot+ 0.3.2

- The autoscaling of logarithmic axes made more trust-worthy: error checks to ensure that they do not try to cover negative ordinates.
- Error checks put in place to prevent empty keys being plotted, which made PyX crash previously. Now can plot empty graphs happily.

- Datasets with blank titles removed from the key, to allow users to plot some datasets to be omitted from the key. This is not possible in gnuplot.
- Bug fix to prevent PyX's texrunner from crashing irreparably upon receiving bad LaTeX. Now uses a spanner to attempt to return it to working order for the next plot.
- Bug fix to the autoscaling of axes with no range of data – previous did not work for negative ordinates. Now displays an axes with a range of +/- 1.0 around the data.

2006 Apr 12: GnuPlot+ 0.3.1

- Plotting of functions fixed: plot command will now plot any algebraic expression, not just functions of the form f(x).
- Space added after command prompt.

2006 Apr 12: GnuPlot+ 0.3.0

- X11_singlewindow and X11_multiwindow terminals implemented, as distinct from just standard X11.
- Key positioning allowed to be xcentre, ycentre, below and outside, as well as in the corners of the plot. Key allowed to be offseted in position.
- Datasets colours can be set via 'plot with colour <n>'
- Datasets are split when there is a blank line in the datafile; lines are not joined up between the two segments.
- set size implemented; can now change aspect ratio of plots.
- working directory of GnuPlot+ changed to /tmp, so that LaTeX's temporary files are stored there rather than in the user's cwd.

2006 Mar 30: GnuPlot+ 0.2.0

- Standard GnuPlot dual axes improved upon, allowing users to add x3, x4 axes, etc, up to any number of axes that may be desired.
- Autocomplete mechanism for commandline substantially cleaned up and debugged.
- Bug fixes to the plotting of arrows/labels. Now appear *above* gridlines, not below.

2006 Feb 26: GnuPlot+ 0.1.0

Index