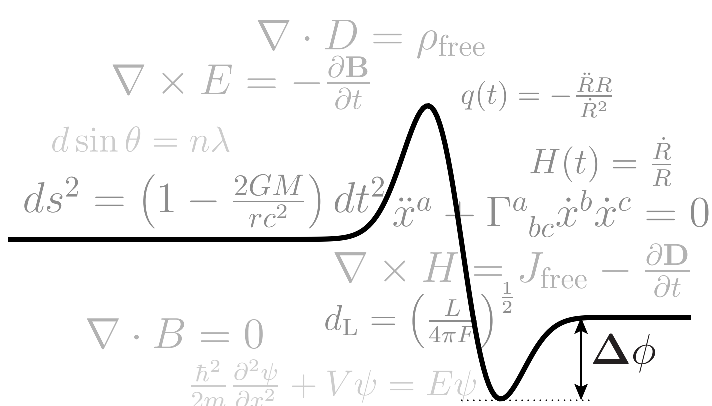
PyXPlot Users' Guide

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

Version 0.6.3



Dominic Ford

Email: `coders@pyxplot.org.uk`

February 2007

Contents

Chapter 1

Introduction

1.1 Overview

PyXPlot is a command-line 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 command-line 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 on the project website[1].

As well as the ease of use and flexibility of gnuplot's command-line 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

[1] `http://www.pyxplot.org.uk/`

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 2.11. 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 (see Section 2.3 for more details). For example:

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

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

```
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. For users with no prior knowledge of LaTeX the author recommends Tobias Oetiker's excellent introduction, *The Not So Short Guide to LaTeX2ε*[2].

1.2 System Requirements

PyXPlot works on many UNIX-like operating systems. The authors have tested it under Linux, SunOS and MacOS X, and believe that it should work on other similar systems. It requires that the following software packages (not included) be installed:

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

python	(Version 2.3 or later)
latex	(Used for all textual labels)
convert	(ImageMagick; needed for the gif, png and jpg terminals)

The following package is not required for installation, but many PyXPlot features are disabled when it is not present, including the `fit` and `spline` commands and the integration of functions. It is very strongly recommended:

scipy (Python Scientific Library)

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/Ubuntu users can find the above software in the packages `tetex-extra`, `gv`, `imagemagick`, `python2.3`, `python2.3-scipy`.

1.3 Installation

1.3.1 Installation as User

The following steps describe the installation of PyXPlot from a `.tar.gz` archive by a user without superuser (i.e. root) access to his machine. It is assumed that the packages listed above have already been installed; if they are not, you need to contact your system administrator.

- Unpack the distributed .tar.gz:

  ```
  tar xvfz pyxplot_0.6.3.tar.gz
  cd pyxplot
  ```

- Run the installation script:

  ```
  ./configure
  make
  ```

- Finally, start PyXPlot:

  ```
  ./pyxplot
  ```

1.3.2 System-wide Installation

Having completed the steps described above, PyXPlot may be installed system-wide by a superuser with the following additional step:

```
make install
```

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.

PyXPlot may now be started by any system user, simply by typing:

```
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 2.3, describing the use of LaTeX to render text, and Section 2.11, 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 command-line 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 command line, 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 command line, 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 command line:

```
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) =
```

2.3 Axis Labels and Titles

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)
```

The output from this simple example is shown in Figure 2.1. All such text labels should be placed between either single (') or double (") quotes. They are displayed using LaTeX, and so any LaTeX commands 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 ~.

Because of the use of quotes to delimit text labels, special care needs to be taken when apostrophe and quote characters are used. The following command would raise an error, because the apostrophe would be interpreted as marking the end of the text label:

```
set xlabel 'My plot's X axis'
```

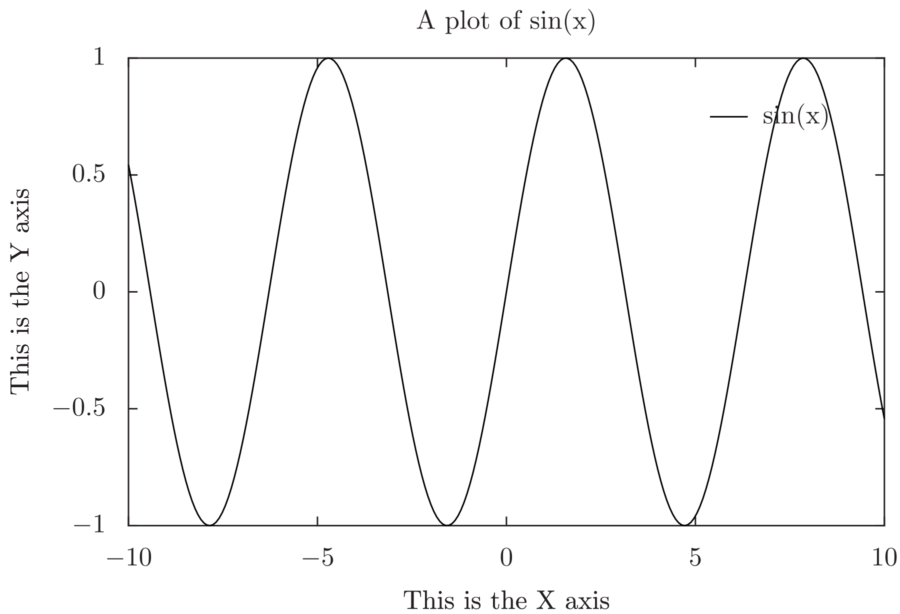


Figure 2.1: A simple first plot with axis labels. See the text for more details.

The following would achieve the desired effect:

```
set xlabel "My plot's X axis"
```

To allow one to render LaTeX strings containing both single and double quote characters – for example, to put a German umlaut on the name "Jörg" in the LaTeX string "`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. To render the example string above, one would type:

```
set xlabel "J\\"org's Data"
```

Two backslashes are used. The first backslash is the LaTeX escape character used to produce the umlaut; the second is a PyXPlot escape character, used so that the " character is not interpretted as delimiting the string.

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.

2.4 Operators and Functions

As has already been seen above, some mathematical functions such as $\sin(x)$ are pre-defined within PyXPlot. A list of frequently-used functions which are predefined in PyXPlot is given in Table 2.2[2]. A list of operators recognised by PyXPlot is given in Table 2.4.

2.5 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:

[2] Users with some experience in Python may be interested to know that all of the functions in the core and math modules are recognised.

acos(x)	Return the arc cosine (measured in radians) of x.
asin(x)	Return the arc sine (measured in radians) of x.
atan(x)	Return the arc tangent (measured in radians) of x.
atan2(y, x)	Return the arc tangent (measured in radians) of y/x. Unlike atan(y/x), the signs of both x and y are considered.
ceil(x)	Return the ceiling of x as a float. This is the smallest integral value $\geq x$.
cos(x)	Return the cosine of x (measured in radians).
cosh(x)	Return the hyperbolic cosine of x.
degrees(x)	Convert angle x from radians to degrees.
exp(x)	Return e raised to the power of x.
fabs(x)	Return the absolute value of the float x.
floor(x)	Return the floor of x as a float. This is the largest integral value $\leq x$.
fmod(x, y)	Return fmod(x, y), according to platform C. x % y may differ.
frexp(x)	Return the mantissa and exponent of x, as pair (m, e). m is a float and e is an int, such that $x = m \times 2^e$. If x is 0, m and e are both 0. Else $0.5 \leq \mathrm{abs}(m) < 1.0$.
hypot(x, y)	Return the Euclidean distance, $\sqrt{x^2 + y^2}$.
ldexp(x, i)	Return $x \times 2^i$.
log($x[, base]$)	Return the logarithm of x to the given base. If the base not specified, returns the natural logarithm (base e) of x.
log10(x)	Return the base 10 logarithm of x.
max(x,y,...)	Return the greatest of the numerical values supplied.
min(x,y,...)	Return the least of the numerical values supplied.
modf(x)	Return the fractional and integer parts of x. Both results carry the sign of x. The integer part is returned as a real.
pow(x, y)	Return x^y.
radians(x)	Converts angle x from degrees to radians.
sin(x)	Return the sine of x (measured in radians).
sinh(x)	Return the hyperbolic sine of x.
sqrt(x)	Return the square root of x.
tan(x)	Return the tangent of x (measured in radians).
tanh(x)	Return the hyperbolic tangent of x.

Table 2.2: A list of mathematical functions which are pre-defined within PyXPlot.

`+`	Algebraic sum
`-`	Algebraic subtraction
`*`	Algebraic multiplication
`**`	Algebraic exponentiation
`/`	Algebraic division
`%`	Modulo operator
`<<`	Left binary shift
`>>`	Right binary shift
`&`	Binary and
`|`	Binary or
`^`	Logical exclusive or
`<`	Magnitude comparison
`>`	Magnitude comparison
`<=`	Magnitude comparison
`>=`	Magnitude comparison
`==`	Equality comparison
`!=`	Equality comparison
`<>`	Alias for `!=`
`and`	Logical and
`or`	Logical or

Table 2.4: A list of mathematical operators which PyXPlot recognises.

`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 2.2: An example PyXPlot datafile – the datafile is shown in the two left-hand columns, and commands are shown to the right.

```
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 3.4, 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 2.2.

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.

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 3.4.

2.6 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 graphical formats. The `set terminal` command[3] 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:

[3]gnuplot users should note that the syntax of the `set terminal` command in PyXPlot is rather different; see Section 3.2.

```
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. The former is recommended for producing figures to embed into documents, the latter for plots which are to be printed without further processing. The `postscript` terminal produces the latter; the `eps` terminal should be used to produce the former. Similarly the `pdf` terminal produces pdf files:

```
set terminal postscript
set terminal eps
set terminal pdf
```

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 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.

2.7 Data Styles

By default, data from files are 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[4]:

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

```
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 2.5), the `using` modifier can be used:

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

More details of the errorbars plot style can be found in Section 3.4. Other plots styles supported by PyXPlot are listed in Section 2.11, 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.

The default plotting style referred to above can also be changed. For example:

```
set style data lines
```

The default style for plotting data from files is then changed to lines. Similarly the "`set style function`" command changes the default style for plotting functions.

2.8 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]`.[5] The first specified range applies to the x-axis, and the second to the y-axis.[6] Any of the values 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.

```
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.

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

[6] 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.

2.9 Function Fitting

It is possible to fit functional forms to data points in datafiles using the `fit` command. A simple example might be:[7]

```
f(x) = a*x+b
fit f() '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.[8]

This is useful for producing best-fit lines[9], 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 3.4) 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 $\{a = 100, b = 50\}$:

  ```
  f(x) = a*x+b
  a = 100
  b = 50
  fit f() '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:

[7] In gnuplot, this example would have been written `fit f(x) ...`, rather than `fit f() ...`. This syntax is supported in PyXPlot, but deprecated.

[8] The `select` keyword, to be introduced in Section 3.4 can also be used.

[9] 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
fit f() '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() '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 2.8. 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.

2.10 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.

2.11 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.[10]
- 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 3.4 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 chapter.
- Placing arrows and textual labels on plots.
- 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).

[10] 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 "’"‘`.

- Setting major/minor tics with the `set xtics` and `set mxtics` commands.

Gnuplot features which PyXPlot does not presently support include:

- Parametric function plotting.
- Three-dimensional plotting (i.e. gnuplot's `splot` command).

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 command-line 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 command line, the PyXPlot accepts the following switches which modify its behaviour:

`-h --help`	Display a short help message listing the available command-line 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.

If the `enlarge` modifier is used with the `set terminal` command then the whole plot is enlarged or, in the case of large plots, shrunk to the current paper size (minus a small margin). The aspect ratio of the plot is preserved. This is perhaps most useful when preparing a plot to send to a printer with the postscript terminal.

In Section 2.6 I described how the `set terminal` command can be used to produce plots in various graphic formats. In addition, I here describe

[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 colourblind, 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.

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.

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 an additional option, the `X11_persist` terminal keeps plot windows open after PyXPlot exits; the above two terminals close all plot windows upon exit.

As there are many changes to the options accepted by the `set terminal` command in comparison to those understood by gnuplot, the syntax of PyXPlot's command is given below, followed by a list of the recognised settings:

```
set terminal { X11_singlewindow | X11_multiwindow | X11_persist |
               postscript | eps | pdf | gif | png | jpg }
             { colour | color | monochrome }
             { portrait | landscape }
             { invert | noinvert }
             { transparent | solid }
             { enlarge | noenlarge }
```

`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**]

[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.

`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.
`x11_persist`	As above, but plot windows remain open after PyX-Plot closes.
`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`	As above, but produces encapsulated postscript.
`pdf`	As above, but produces pdf output.
`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.
`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.
`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.
`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**]
`enlarge`	Enlarge or shrink contents to fit the current paper size.
`noenlarge`	Do not enlarge output; opposite to the above. [**default**]

The default terminal is normally `x11_singlewindow`, matching approx-

[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).

imately the behaviour of gnuplot. However, there is an exception to this. When PyXPlot is used non-interactively – i.e. one or more command scripts are specified on the command line, 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 command line, 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 this 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).

3.2.1 Paper Sizes

By default, when the postscript terminal produces printable, i.e. not encapsulated, output, the paper size for this output is read from your system locale settings. It may be changed, however, with the `set papersize` command, which may be followed either by the name of a recognised paper size, or by the dimensions of a user-defined size, specified as a `height`, `width` pair, both being measured in millimetres. For example:

```
set papersize a4
set papersize 100,100
```

A list of recognised paper size names is given in Figure 3.1.

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:

Name	h/**mm**	w/**mm**	**Name**	h/**mm**	w/**mm**
2a0	1681	1189	medium	584	457
4a0	2378	1681	monarch	267	184
a0	1189	840	post	489	394
a1	840	594	quad_demy	1143	889
a10	37	26	quarto	254	203
a2	594	420	royal	635	508
a3	420	297	statement	216	140
a4	297	210	swedish_d0	1542	1090
a5	210	148	swedish_d1	1090	771
a6	148	105	swedish_d10	48	34
a7	105	74	swedish_d2	771	545
a8	74	52	swedish_d3	545	385
a9	52	37	swedish_d4	385	272
b0	1414	999	swedish_d5	272	192
b1	999	707	swedish_d6	192	136
b10	44	31	swedish_d7	136	96
b2	707	499	swedish_d8	96	68
b3	499	353	swedish_d9	68	48
b4	353	249	swedish_e0	1241	878
b5	249	176	swedish_e1	878	620
b6	176	124	swedish_e10	38	27
b7	124	88	swedish_e2	620	439
b8	88	62	swedish_e3	439	310
b9	62	44	swedish_e4	310	219
c0	1296	917	swedish_e5	219	155
c1	917	648	swedish_e6	155	109
c10	40	28	swedish_e7	109	77
c2	648	458	swedish_e8	77	54
c3	458	324	swedish_e9	54	38
c4	324	229	swedish_f0	1476	1044
c5	229	162	swedish_f1	1044	738
c6	162	114	swedish_f10	46	32
c7	114	81	swedish_f2	738	522
c8	81	57	swedish_f3	522	369
c9	57	40	swedish_f4	369	261
crown	508	381	swedish_f5	261	184
demy	572	445	swedish_f6	184	130
double_demy	889	597	swedish_f7	130	92
elephant	711	584	swedish_f8	92	65
envelope_dl	110	220	swedish_f9	65	46
executive	267	184	swedish_g0	1354	957
foolscap	330	203	swedish_g1	957	677
government_letter	267	203	swedish_g10	42	29
international_businesscard	85	53	swedish_g2	677	478
japanese_b0	1435	1015	swedish_g3	478	338
japanese_b1	1015	717	swedish_g4	338	239
japanese_b10	44	31	swedish_g5	239	169
japanese_b2	717	507	swedish_g6	169	119
japanese_b3	507	358	swedish_g7	119	84
japanese_b4	358	253	swedish_g8	84	59
japanese_b5	253	179	swedish_g9	59	42
japanese_b6	179	126	swedish_h0	1610	1138
japanese_b7	126	89	swedish_h1	1138	805
japanese_b8	89	63	swedish_h10	50	35
japanese_b9	63	44	swedish_h2	805	569
japanese_kiku4	306	227	swedish_h3	569	402
japanese_kiku5	227	151	swedish_h4	402	284
japanese_shiroku4	379	264	swedish_h5	284	201
japanese_shiroku5	262	189	swedish_h6	201	142
japanese_shiroku6	188	127	swedish_h7	142	100
large_post	533	419	swedish_h8	100	71
ledger	432	279	swedish_h9	71	50
legal	356	216	tabloid	432	279
letter	279	216	us_businesscard	89	51

Figure 3.1: A list of all of the named paper sizes recognised by the `set papersize` command, with their heights, h, and widths, w, measured in millimetres.

```
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. This is illustrated in Figure 3.2.

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 equivalent ranges 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

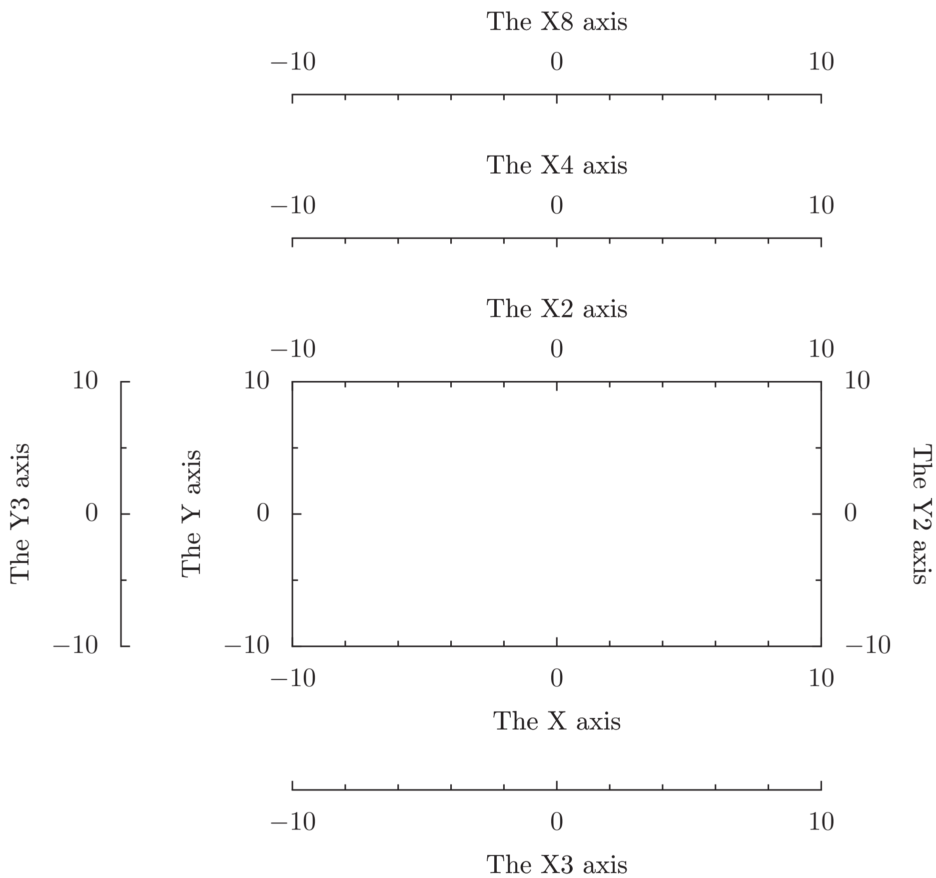


Figure 3.2: A plot demonstating the use of large numbers of axes. Odd-numbered x-axes appear below the plot, and even numbered x-axes above it; a similar rule applies for y-axes, to the left and to the right.

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’.

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 3.5.2.

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
```

The position of ticks along each axis can be configured with the `set xtics` command. The appearance of ticks along any axis can be turned off with the `set noxtics` command. The syntax for these is given below:

```
set xtics { axis | border | inward | outward | both }
          {  autofreq
           | <increment>
           | <minimum>, <increment> { , <maximum> }
           | (     {"label"} <position>
               { , {"label"} <position> } .... )
          }
set noxtics
show xtics
```

The keywords `inward`, `outward` and `both` alter the directions of the ticks, and have the same effect as in the `set xticdir` command. The keyword `axis` is an alias for `inward`, and `border` an alias for `outward`, both provided for gnuplot compatibility. If the keyword `autofreq` is given, the automatic placement of ticks on the axis is restored.

If `<minimum>, <increment>, <maximum>` are specified, then ticks are placed at evenly spaced intervals between the specified limits. In the case of logarithmic axes, `<increment>` is applied multiplicatively.

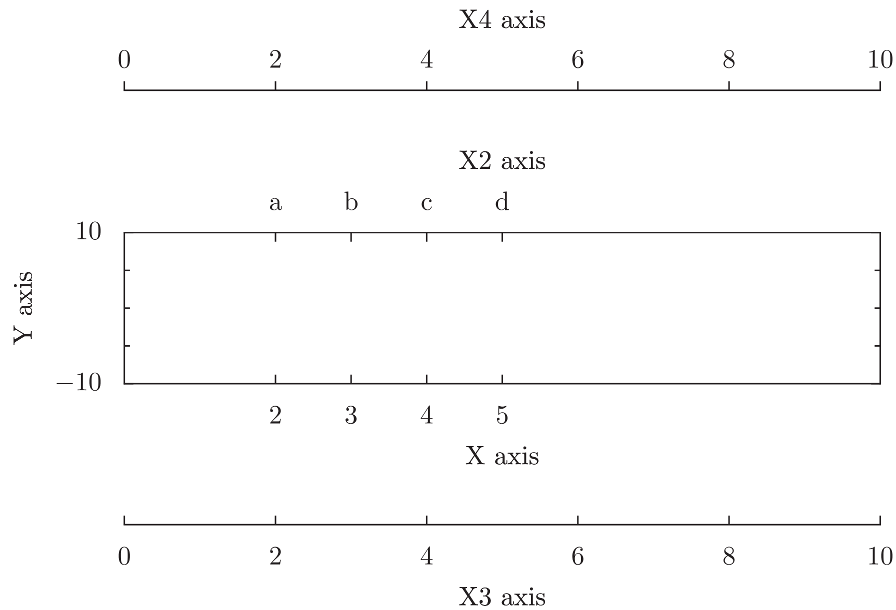


Figure 3.3: A plot demonstrating the use of the `set xtics` command. The commands used to create the axes in this plot are as given in the text.

Alternatively, the final form allows ticks to be placed on an axis individually, and each given its own textual label.

The following pair of examples would both place tick marks at $x = 2$, 3, 4, 5. In the second example, they would be labelled "a", "b", "c" and "d":

```
set xtics 2, 1, 5

set x2tics ("a" 2, "b" 3, "c" 4, "d" 5)
```

The following example would place tick marks at intervals of two units along the $x3$-axis:

```
set x3tics 2
```

The following example would restore the automatic placement of ticks along the $x4$-axis, placing those ticks facing outwards from the graph:

```
set x4tics border autofreq
```

All of the examples above are illustrated in Figure 3.3. Minor tick marks can be placed on axes with the `set mxtics` command, which has the same syntax as above.

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
```

An example of a plot with a two-column legend is given in Figure 3.4.

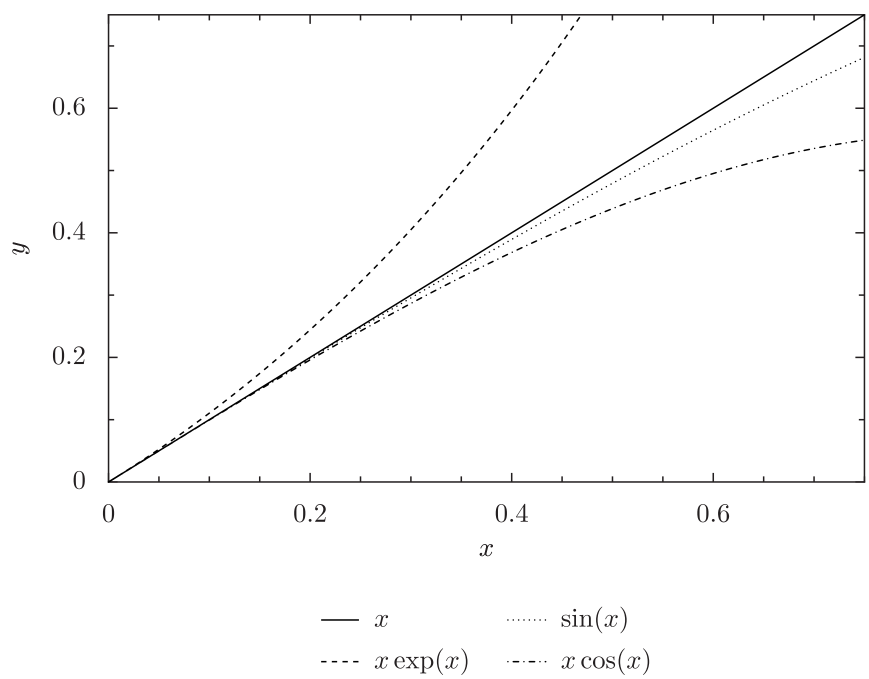


Figure 3.4: An example of a plot with a two-column legend, positioned below the plot using `set key below`.

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 permitted 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))
```

In the second example, two select criteria are given, combined with the logical and operator[6]. 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.

[6] See Table 2.4 for a list of all operators recognised by PyXPlot.

arrows plot style — The `arrows` plot style 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[7] 12 and 13 respectively, obtained as follows:

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

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
```

[7] The `pointtype` modifier was introduced in Section 2.7.

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 2.5. 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 2.5) 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 command line 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 tilde 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. A rotation angle may optionally be specified after the keyword '`rotate`', to rotate text counter-clockwise by a given angle, measured in degrees. For example, the following would produce upward-running text:

```
set label 1 'Hello World' at axis3 3.0, axis4 2.7 rotate 90
```

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.

3.5.1 Deleting, Moving and Changing Plots

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 to 8,8
```

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.

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.

3.5.2 Linked Axes

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.

3.5.3 Text Labels, Arrows and Images

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' at x,y
```

In this case, the string "This is some text" would be rendered at position (x, y) on the multiplot. As with the `set label` command, a rotation angle may optionally be specified to rotate text labels through any given angle, measured in degrees counter-clockwise, for example:

```
text 'This is some text' at x,y rotate r
```

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 with the `delete`, `undelete` and `move` commands.

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. One obvious application is to produce equations rendered as graphical files for inclusion in talks.

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.

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
```

Bitmap images in jpeg form may be placed on the multiplot using the `jpeg` command. This has syntax:

```
jpeg 'filename' at x,y width w
```

As an alternative to the `width` modifier the height of the image can be specified, using the analogous `height` modifier. An optional angle can also be specified using the `rotate` modifier; this causes the included image to be rotated counter-clockwise by a specified angle (in degrees).

Vector graphic images in eps format may be placed on to a multiplot using the `eps` command, which has a syntax analogous to the `jpeg` command. However neither height nor width need be specified; in this case the image will be included at its native size. For example:

```
eps 'filename' at 3,2 rotate 5
```

will place the eps file with its bottom-left corner at position $(3, 2)$ cm from the origin, rotated counter-clockwise through 5 degrees.

3.5.4 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
```

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 midpoints along the x-axis between the specified datapoints (see, for example, Panel (a) of Figure 3.5). Alternatively, the widths of the bars may be set

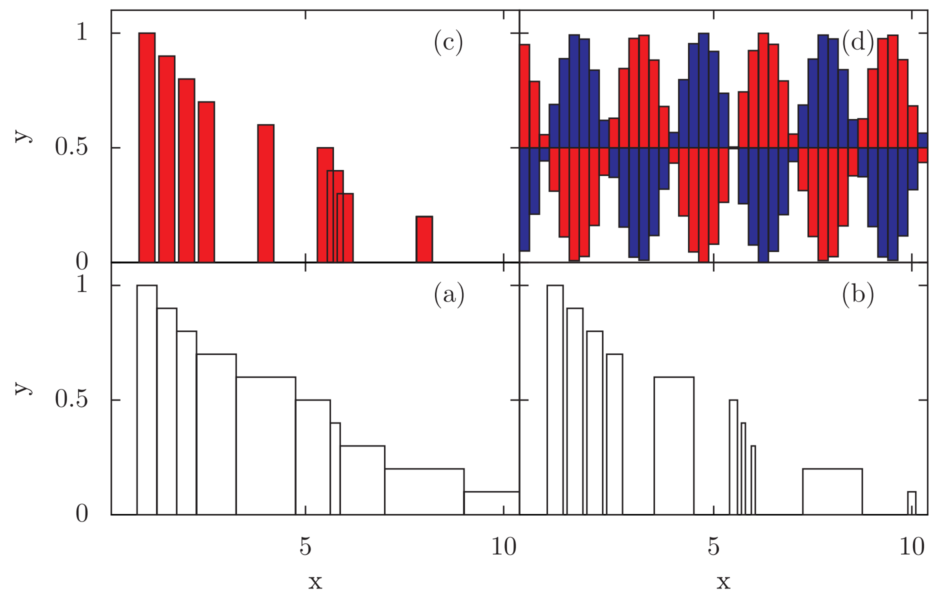


Figure 3.5: A gallery of different bar chart styles which PyXPlot can produce. See the text for more details.

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 3.5.

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 3.5 shows an example of this plot style in use.

By default, the bars all originate from the line $y = 0$, as is normal 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
```

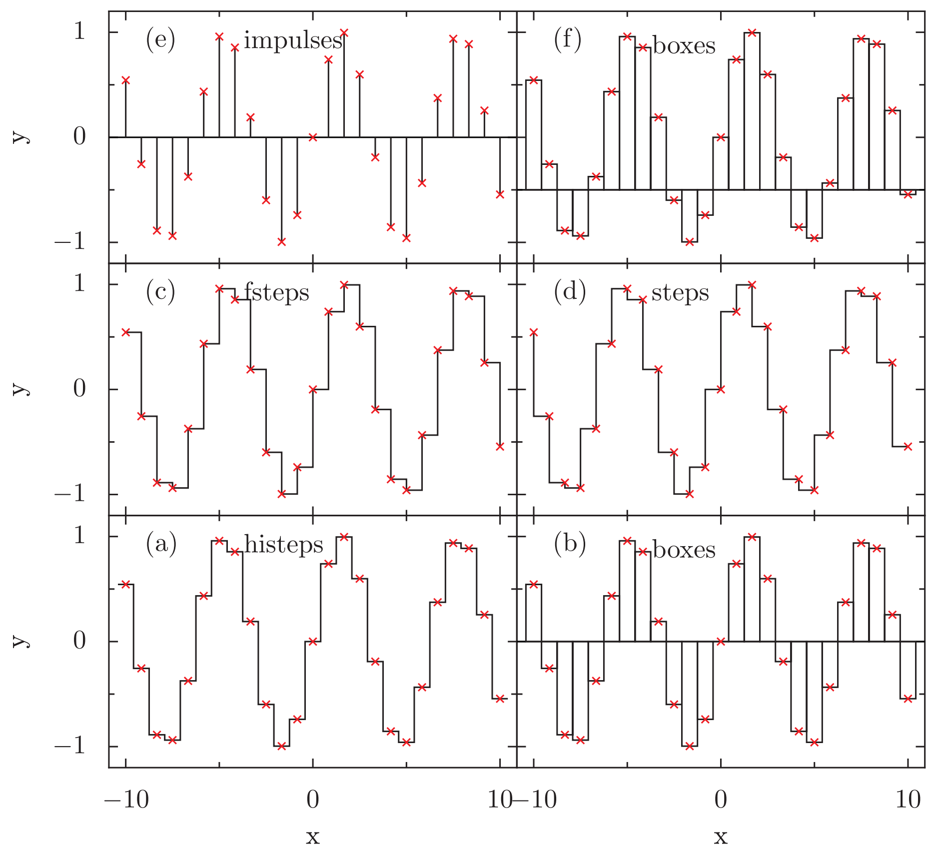


Figure 3.6: A second gallery of different bar chart styles which PyXPlot can produce. See the text for more details.

All of the bars would then originate from the line $y = 5$. Panel (f) of Figure 3.6 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
```

Panels (c) and (d) of Figure 3.5 demonstrate the use of filled bars.

Finally, the `impulses` plot style, as in gnuplot, produces bars of zero width; see Panel (e) of Figure 3.6 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 3.6 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 command line:

```
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 $-2 < x < 2$:[8]

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

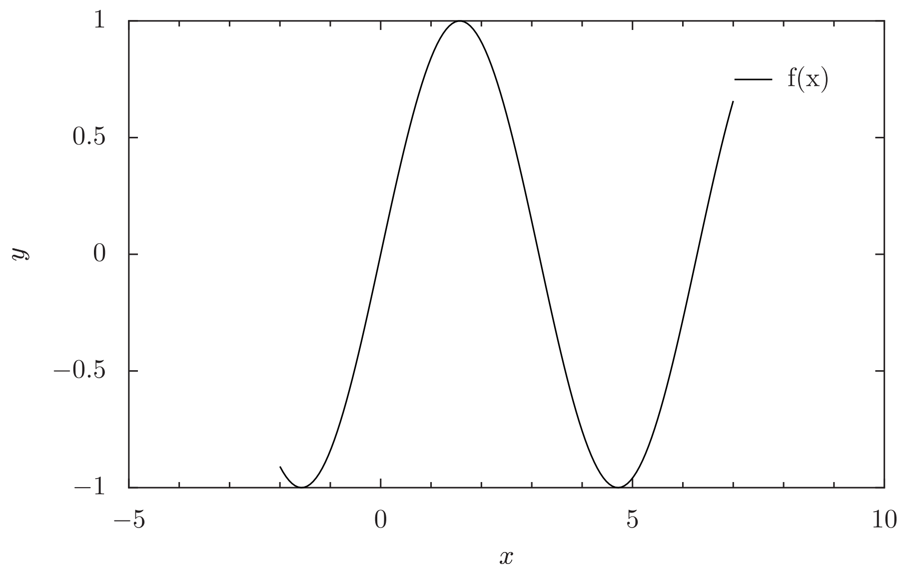


Figure 3.7: A simple example of the use of function splicing to truncate the function $\sin(x)$ at $x = -2$ and $x = 7$. See details in the text.

```
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 $-2 < x < 7$:

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

The output of this particular example can be seen in Figure 3.7. A similar effect could also have been achieved with the `select` keyword; see Section 3.4.

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.

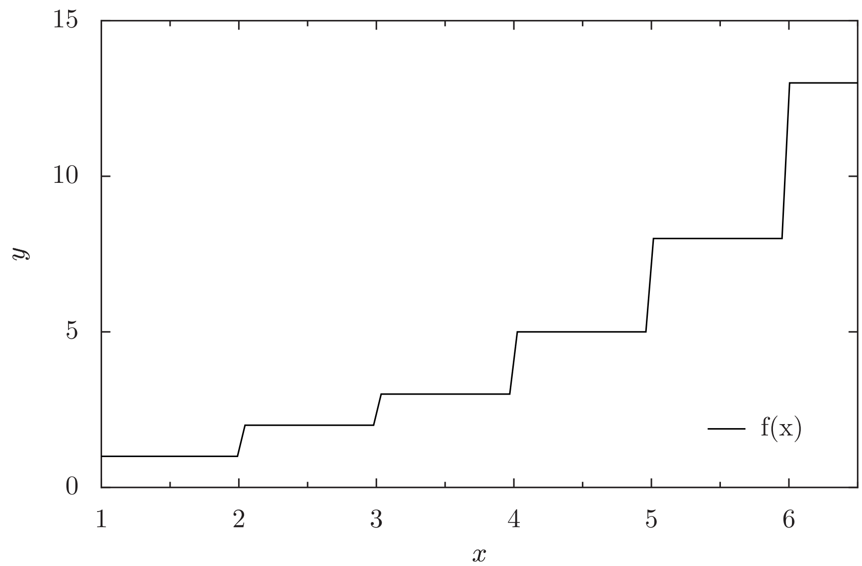


Figure 3.8: An example of the use of function splicing to define a function which does not have an analytic form – in this case, the Fibonacci sequence. See the text for details.

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 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)
```

The output of this example can be seen in Figure 3.8

3.8 Datafile Interpolation: Spline Fitting

The `spline` command fits a spline through data from a file. 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 gnuplot `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 obligatory parameters plus two further optional parameters. The first is the expression to be differentiated, followed by the point at which the differential should be evaluated, following by optional parameters ϵ_1 and ϵ_2, which are described below. 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 a parameter ϵ controls the spacing with which to perform the differencing operation:

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

where $\epsilon = \epsilon_1 + x\epsilon_2$. By default, $\epsilon_1 = \epsilon_2 = 10^{-6}$, which is appropriate for the differentiation of most well-behaved functions.

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 command-line 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 command line, they are watched and executed independently, *not* sequentially, as PyXPlot itself would do. Wildcard characters can also be used to set `pyxplot_watch` to watch multiple files.[9]

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
```

[9]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. The preamble can be changed using the `set preamble` 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
ENLARGE=OFF
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 3.4).
`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 3.5.4 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).
`ENLARGE`	**Possible values:** ON / OFF **Analogous set command:** `set terminal` When set to 'ON' output is enlarged or shrunk to fit the current paper size.
`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.
`PAPER_HEIGHT`	**Possible values:** Any floating-point number. **Analogous set command:** `set papersize` Sets the height of the papersize for postscript output in millimetres.
`PAPER_WIDTH`	**Possible values:** Any floating-point number. **Analogous set command:** `set papersize` Sets the width of the papersize for postscript output in millimetres.
`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`, `X11_persist`, `PS`, `EPS`, `PDF`, `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 command-line 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 command-line 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 command-line 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 command-line switches:** None. As above, but sets the colour in which PyXPlot displays its warning messages.
`SPLASH`	**Possible values:** ON / OFF **Analogous command-line 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 common 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

Command Reference

This chapter contains an alphabetically ordered list of all the commands that PyXPlot understands.

5.1 arrow

```
arrow [from] <x>, <y> [to] <x>, <y> [with <option> ... ]
```

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

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

The arrow command may be followed by the '`with`' keyword to specify to style of the arrow. The style keywords which are accepted are '`nohead`', '`head`' (default) or '`twohead`', in addition to keywords such as '`colour`', '`linewidth`' or '`linetype`', which have the same syntax and meaning as they do in the plot command. An example would be:

```
arrow from x,y to x,y with twohead linetype 2 colour blue
```

Arrows receive unique identification numbers which count sequentially from one, and which are output to the terminal after the arrow command is called. By reference to these numbers, they can later be deleted and undeleted with the `delete` and `undelete` commands respectively. For example:

```
delete 2
```

5.2 cd

```
cd <directory>
```

PyXPlot's `cd` command is very similar to the shell cd command; it can be used to change the current working directory. For example:

```
cd foo
```

5.3 clear

```
clear
```

In multiplot mode the `clear` command removes all current plots, arrows and text objects from the working page. In single plot mode it is not especially useful; it removes the current plot to leave a blank page.

The `clear` command should not be followed by any parameters.

5.4 delete

```
delete <plot number>, ...
```

The `delete` command is part of the multiplot environment; it removes plots, arrows or text items from a multiplot page. The desired items should be identified using a comma-separated list of their reference numbers, which count sequentially from zero for the first item created on a multiplot page, and are displayed on the terminal when items are created. For example:

```
delete 1,2,3
```

removes item numbers 1, 2 and 3.

Having been deleted, multiplot items can be restored using the `undelete` command.

5.5 edit

```
edit <plot number>
```

The `edit` command is part of the multiplot environment; it allows one to modify the properties of any plot on a multiplot. The desired plot should be identified using the reference number which it was given when it was created using the `plot` command; it would have been displayed on the terminal at that time. For example, consider the following command sequence:

```
edit 1
set textcolour red
replot
```

Here, the `edit` command sets the following `set textcolour` command to affect the plot with reference number 1 – the first plot which would have been placed on the multiplot. The `set textcolour red` command then affects this plot, although does not take effect until the `replot` command is called.

The `edit` command also has the effect of resetting all of PyXPlot's plot settings to those used to produce the chosen plot, and so in conjunction with the `show` command, can be used to inspect as well as modify the settings of any plot on a multiplot page. For example:

```
edit 1
show textcolour
```

would show the text colour used in plot 1.

Having issued the `edit` command, no further command needs to be issued to return to a state of adding plots to a multiplot rather than editing the existing plots; simply call the `plot` command rather than the `replot` command to do this.

5.6 eps

```
eps '<filename>' [at <x>, <y>] [rotate <angle>] [width <width>]
                 [height <height>]
```

The `eps` command places an eps format file into the current plot. The `at` modifier can be used to specify the position of the bottom-left corner of the file, otherwise it is placed at the origin. If the `rotate` modifier is used the image is rotated by the specified angle counter-clockwise. Either the `width` or `height` can be specified, followed by the width or height respectively in cm that the resulting image should be; otherwise the native file width will be used. The `eps` command is perhaps most useful in multiplot mode, where included files can be combined with plots, text labels, etc.

5.7 exit

```
exit
```

The `exit` command can be used to quit PyXPlot. If multiple command files, or a mixture of command files and interactive sessions, are specified on the command line, then PyXPlot moves onto the next command-line item after receiving the `exit` command.

PyXPlot may also be quit be pressing CTRL-D or via the `quit` command. In interactive mode, CTRL-C terminates the current command, if one is running. When running a script, CTRL-C terminates execution of it.

5.8 fit

```
fit [<range specifier> ...] <function> '<datafile>'
    [index <index specifier>] [using <using specifier>]
    via <variable>[, <variable>, ...]
```

The `fit` command may be used to fit functional forms to data points in datafiles. 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.

This is useful for producing best-fit lines and also has applications for estimating the gradients of datasets. The syntax is essentially identical to the used by gnuplot, though a few points are worth noting, which are outlined in Section 2.9.

5.9 help

```
help [<topic> [<sub-topic> ... ] ]
```

The `help` command provides an easily-navigable source of information which is supplementary to that in this manual. To obtain information on any particular topic, type `help` followed by the name of the topic. For example:

```
help commands
```

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

```
help commands help
```

Information is arranged with general information about PyXPlot under the heading `about`, and information about PyXPlot's commands under `commands`. Information about the format that input datafiles should take can be found under `datafile`. Other categories are self-explanatory.

To exit any help page, press the '`q`' key.

5.10 jpeg

```
jpeg '<filename>' [at <x>, <y>] [rotate <angle>] [width <width>]
                  [height <height>]
```

The `jpeg` command places a jpeg format bitmap image into the current plot. The `at` modifier can be used to specify the position of the bottom-left corner of the image, otherwise it is placed at the origin. If the `rotate` modifier is used the image is rotated by the specified angle counter-clockwise. Either the `width` or `height` modifier should be specified, followed by the width or height respectively in cm that the resulting image should be. The `jpeg` command is perhaps most useful in multiplot mode, where images can be combined with plots, text labels, etc.

5.11 load

```
load '<filename>'
```

The `load` command executes a PyXPlot command script file, just as if its contents had been typed into the current terminal. For example:

```
load 'foo'
```

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

Wildcards can be used in the load command, in which case *all* commandfiles matching the given wildcard are executed, for example:

```
load '*.script'
```

5.12 move

```
move <plot number> to <x>, <y>
```

The `move` command is part of the multiplot environment; it can be used to move items around on a multiplot page. The desired item to be moved should be identified using the reference number which it was given when it was created; it would have been displayed on the terminal at that time. For example:

```
move 23 to 8,8
```

This would move multiplot item 23 to position 8,8 (measured in centimetres). If this item were a plot, the end result would be the same as if the command `set origin 8,8` had been called before it had originally been plotted.

5.13 !

```
! <shell command>
<command> `<shell command>` ...
```

Shell commands can be executed from within PyXPlot by pre-fixing them with pling (!) characters, for example:

```
!mkdir foo
```

As an alternative, back-quotes (`) can be used to substitute the output of a shell command into a PyXPlot command, for example:

```
set xlabel `echo "'" ; ls ; echo "'"`
```

Note that back-quotes cannot be used inside quote characters, and so the following would *not* work:

```
set xlabel '`ls`'
```

5.14 plot

```
plot [<range specifier> ...] ('<filename>'|<function>)
     [using <using specifier>] [axes <axis specifier>]
     [select <select specifier>]
     [index <index specifier>]
     [every <every specifier>]
     [with <style> [<style modifier> ... ] ]
```

The `plot` command is the main workhorse command of PyXPlot, which is used to produce all plots. For example to plot the sine function:

```
plot sin(x)
```

Ranges for the axes of a graph can be specified by placing them in square-brackets before the name of the function to be plotted. Leaving a set of brackets empty specifies that an axis will be automatically scaled, as happens by default. An example of this syntax would be:

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

which would plot the function $\sin(x)$ across some default range of values on the x-axis.

Datafiles may also be plotted as well as functions, in which case the filename of the datafile to be plotted should be enclosing in apostrophes. An example of this syntax would be:

```
plot 'datafile' with points
```

which would plot the file called '`datafile`'. Section 2.5 should be studied for further details of the format that is expected of input datafiles, and how PyXPlot may be directed to plot only certain portions of datafiles.

In plots which have multiple parallel axes – for example, an x-axis along its lower edge and an $x2$-axis along its upper edge – the pair of axes against which data should be plotted should be specified using the modifier `axes` following the name of the function or datafile to be plotted, for example:

```
plot sin(x) axes x2y1
```

The style in which data should be plotted may be specified following the modifier `with`, with the following syntax:

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

The following plot styles are recognised: `lines`, `points`, `linespoints`, `dots`, `boxes`, `wboxes`, `impulses`, `steps`, `histeps`, `fsteps`, `xerrorbars`, `yerrorbars`, `xyerrorbars`, `xerrorrange`, `yerrorrange`, `xyerrorrange`, `arrows_head`, `arrows_nohead`, `arrows_twohead`, `csplines`, `acsplines`.

In addition, `lp` and `pl` are recognised as abbreviations for `linespoints`; `errorbars` is recognised as an abbreviation for `yerrorbars`; `errorrange` is recognised as an abbreviation for `yerrorrange`; and `arrows_twoway` is recognised as an alternative for `arrows_twohead`.

As well as plot styles, the `with` modifier can also be followed by the following keywords:

`linetype` – specifies the linetype (e.g. dotted) used by the lines plot style.

`linewidth` – specifies the width of line, in pt, used by the lines plot style.

`pointsize` – specifies the size of datapoints, relative to the default size, used by the points plot style.

`pointlinewidth` – as above, but specifies the linewidth, in pt, used to render the crosses, circles, etc, used to mark datapoints.

`linestyle` – this can be used in conjunction with the `set linestyle` command to save default plot styles.

`colour` – specifies the colour used to plot the dataset, either by one of the recognised colour names or by an integer, to use one from the current palette.

`fillcolour` – relavant to the `boxes` and `wboxes` plot styles, specifies a colour in which bar charts should be filled.

An example using several of these keywords would be:

```
plot sin(x) axes x2y1 with colour blue linetype 2 \
                                linewidth 5
```

Multiple datasets can be plotted on a single graph by listing them with commas separating them:

```
plot sin(x) with colour blue, cos(x) with linetype 2
```

5.15 print

```
print <expression>
```

The `print` command outputs the value of a mathematical expression to the terminal. It is most often used to find the value of a variable, though it can also be used to produce formatted output from a PyXPlot script. For example:

```
print a
```

would print the value of the variable a.

5.16 pwd

```
pwd
```

The `pwd` command prints the location of the current working directory.

5.17 ?

```
? [<help option> ... ]
```

The `?` symbol is a shortcut to the `help` command.

5.18 quit

```
quit
```

The `quit` command can be used to exit PyXPlot. If multiple command files, or a mixture of command files and interactive sessions, are specified on the command line, then PyXPlot moves onto the next command-line item after receiving the `exit` command.

PyXPlot may also be quit be pressing CTRL-D or via the `exit` command. In interactive mode, CTRL-C terminates the current command, if one is running. When running a script, CTRL-C terminates execution of it.

5.19 refresh

```
refresh
```

The `refresh` command 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 plot in a different graphic format. It differs from the `replot` command in that it doesn't replot anything; subsequent usages of the `set` command since the previous `plot` command have no affect on the output. The `refresh` command is also especially useful in the multiplot environment; it can be used to produce second copies of multiplot pages where there need not necessarily even be any plots; there might perhaps only be textual items and arrows.

5.20 replot

```
replot [<plot number>]
```

In single plot mode, the `replot` command causes the most recent plot command to be re-run. This can be useful to replot a datafile which has changed in the meantime, but also to change some aspect of a plot within PyXPlot itself. Usages of the `set` command between the original `plot` command and the calling of the `replot` command are applied to the new plot. For example:

```
plot sin(x)
set textcolour red
replot
```

In multiplot mode, the `replot` command acts by default upon the last plot item which was added to the multiplot page, and causes that to be replotted. It is possible to change this behaviour by first calling the `edit` command, in which case any given plot within a multiplot can be modified and replotted.

Specifying a function or datafile after the `replot` command causes that function or data file to be added to the plot. The syntax here is the same as for the `plot` command. For example:

```
replot sin(x) axes x2y1 with linespoints
```

will add a plot of the function $\sin(x)$ to the current plot.

5.21 reset

```
reset
```

The `reset` command returns the values of all settings that have been changed with the `set` command back to their default values.

5.22 save

```
save '<filename>'
```

The `save` command saves a list of all of the commands which have been executed in the current interactive PyXPlot session into a given file. The filename of the desired location for this file should be placed in quotes, for example:

```
save 'foo'
```

would save a command history into the file named '`foo`'.

5.23 set

```
set <option> <value>
```

The `set` command sets the value of various operational parameters within PyXPlot. For example:

```
set pointsize 2
```

would sets the default point size to 2. The basic syntax always follows that above: the `set` command should be followed by some keyword specifying which setting it is which should be set. If a further parameter is needed to specify what value to set this setting to, it should follow this keyword. Settings which work in an on/off fashion tend to take a syntax along the lines of:

`set key`	Set option ON
`set nokey`	Set option OFF

More details of the functions of each individual setting can be found in the subsections below, which represents a complete list of the recognised setting keywords.

The reader should also see the `show` command, which can be used to display the current values of settings, and the `unset` command, which returns settings to their default values. Section 4.2 describes how commonly used settings can be saved into a configuration file.

5.23.1 arrow

```
set arrow <arrow number> from [<co-ordinate>] <x>,
          [<co-ordinate>] <y> to [<co-ordinate>] <x>,
          [<co-ordinate>] <y> [with <modifier> ]

<co-ordinate> = ( first | second | screen | graph |
                  axis<axisnumber>                  )
```

The `set arrow` command causes an arrow to be added to a plot. An example of its syntax would be:

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

which would cause an arrow to be drawn between the points 0,0 and 1,1, as measured on the x and y axes. The tag '1' immediately following the `arrow` keyword is an identification number, and allows the arrow to be removed later with the `unset arrow` command. By default the co-ordinates are measured relative to the first x- and y-axes, but can be specified in a range of coordinate systems. These are specified as follows:

```
set arrow 1 from first 0, second 0 to axis3 1, axis4 1
```

As can be seen, the name of the desired coordinate system precedes the position value in that coordinate system. The coordinate system `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. The syntax `axisn` may also be used, to use the n th x- or y-axis; for example, `axis3` above.

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`'. Normal line type modifiers can also be used here. For example:

```
set arrow 2 from first 0, second 2.5 to axis3 0,
             axis4 2.5 with colour blue nohead
```

5.23.2 autoscale

```
set autoscale <axis>[<axis>... ]
```

The `autoscale` setting causes PyXPlot to choose the scaling for an axis automatically based on the data and/or functions to be plotted against it. As an example of the syntax:

```
set autoscale x1
```

would cause the size of the first x-axis to be scaled to fit the data. Multiple axes can be specified, viz.:

```
set autoscale x1y3
```

5.23.3 axescolour

```
set axescolour <colour>
```

The `axescolour` setting changes the colour of the plot's axes. For example:

```
set axescolour blue
```

changes the axes to be blue. Any of the recognised colour names listed in Section 4.6 can be used.

5.23.4 axis

```
set axis <axis>, ...
```

The command:

```
set axis x2
```

may be used to add a second x-axis to a plot, with default settings. In general, there is no practical reason to use this command, as a second x-axis would implicitly be created anyway by any of the following statements:

```
set x2label 'foo' \\
set x2ticdir outwards \\
plot sin(x) axes x2y1
```

Of more practical use is the '`unset x2`' command, which is used to remove an axis once it has been added to a plot. After executing:

```
set x2label 'foo'
```

for example, the only way to tell PyXPlot to subsequently produce a plot without a second x-axis would be to delete this axis with the following command:

```
unset axis x2
```

Note that in this case, the `unset x2label` command would be sufficent to remove the label ‘foo’ placed on the new axis, but not sufficient to delete the new axis that the `set x2label` command implicitly created. Multiple axes can be deleted in a single `unset axis` statement, for example:

```
unset axis x2x4x5
```

In the special cases of `unset axis x1` or `unset axis y1`, these axes cannot be deleted; a plot must have at least one x- and one y-axis. Instead, the `unset axis` command restores these axes to their default configurations, removing any set titles or ranges that they might have been given.

5.23.5 backup

```
set backup
```

The setting `backup` changes PyXPlot’s behaviour when it detects that a file which it is about to write is going to overwrite an existing file. Whereas by default the existing file would be overwritten by the new one, when the `backup` setting is turned on, it is renamed, placing a tilde at the end of its filename. For example, suppose that a plot were to be written with filename ‘`out.ps`’, but such a file already existed. With the backup setting turned on the existing file would be renamed ‘`out.ps `’ to save it from being overwritten.

The setting may be turned off via `set nobackup`.

5.23.6 bar

```
set bar ( large | small | <barsize> )
```

The `bar` setting changes the size of the bar on the end of the errorbars, relative to the current pointsize. For example:

```
set bar 2
```

sets the bars to be twice the size of the points. The options ‘`large`’ and ‘`small`’ are equivalent to 1 (the default) and 0 (no bar) respectively.

5.23.7 boxfrom

```
set boxfrom <value>
```

The ‘`boxfrom`’ setting alters PyXPlot’s behaviour when plotting bar charts. It changes the horizontal line (vertical point; y-axis value) from which the boxes of bar charts appear to emanate. The default value is zero (i.e. boxes extend from the line of the y-axis). An example of its syntax would be:

```
set boxfrom 2
```

which would make the boxes of a barchart emanate vertically from the line $y = 2$.

5.23.8 boxwidth

```
set boxwidth <width>
```

The '`boxwidth`' setting alters PyXPlot's behaviour when plotting bar charts. It sets the default width of the boxes used, in graph x-axis units. If the specified width is negative then, as happens by default, the boxes have automatically selected widths, such that the interfaces between them occur at the horizontal midpoints between their specified x-positions. For example:

```
set boxwidth 2
```

would set all boxes to be two units wide.

```
set boxwidth -2
```

would set all of the bars to have differing widths, centred upon their specified x-positions, such that their interfaces occur at the horizontal midpoints between them.

5.23.9 data style

See '`set style data`'.

5.23.10 display

```
set [no]display
```

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
```

5.23.11 dpi

```
set dpi <value>
```

When PyXPlot is set to produce bitmapped graphics output, using the `gif`, `jpg` or `png` terminals (see the '`set terminal`' command), the '`dpi`' setting changes how many dots per inch these graphics files are produced with. That is to say, it changes the image resolution of these file formats:

```
set dpi 100
```

sets the output to a resolution of 100 dots per inch. Higher dpi values yield better quality images, but larger file sizes.

5.23.12 fontsize

```
set fontsize <value>
```

The `fontsize` setting changes the size of the fount[1] used to render all text labels which appear on a plot, including keys, axis labels, etc. The value specified should be an integer in the range -4 to 5, corresponding to LaTeX's tiny (-4) and Huge (5) sizes, for example:

```
set fontsize 2
```

The default value is zero, LaTeX's normal fount size. As an alternative, fount sizes can be specified directly in the LaTeX text of labels, for example:

```
set xlabel '\Large This is a BIG label'
```

5.23.13 function style

See '`set style function`'.

5.23.14 grid

```
set [no]grid <axis> ...
```

The `grid` setting controls whether a grid is placed behind a plot or not. Issuing the command:

```
set grid
```

would cause a grid to be drawn with its gridlines connecting to the ticks of the default axes (usually the first x- and y-axes). Conversely, issuing:

[1]This is not a spelling mistake. 'font', by contrast, *would* be a spelling mistake. See the Oxford English Dictionary.

```
unset grid
```

would remove from the plot all gridlines associated with the ticks of any axes. One or more axes can be specified for the `set grid` command; a grid will then be drawn to connect with the ticks of these axes. An example of this syntax would be:

```
set grid x1 y3
```

which would cause gridlines to be drawn from ticks of the first x- and third y-axes.

It is possible, though not always aesthetically very pleasing, to draw gridlines from multiple parallel axes, for example:

```
set grid x1x2x3
```

5.23.15 gridmajcolour

```
set gridmajcolour <colour>
```

The '`gridmajcolour`' setting changes the colour that is used to plot the gridlines (see the `set grid` command) which are associated with the major ticks of axes (i.e. major gridlines). For example:

```
set gridmajcolour purple
```

would cause the major grid lines to be drawn in purple. Any of the recognised colour names listed in Section 4.6 can be used.

See also the `set gridmincolour` command.

5.23.16 gridmincolour

```
set gridmincolour <colour>
```

The `gridmincolour` setting changes the colour that is used to plot the gridlines (see the `set grid` command) which are associated with the minor ticks of axes (i.e. minor gridlines). For example:

```
set gridmincolour purple
```

would cause the minor grid lines to be drawn in purple. Any of the recognised colour names listed in Section 4.6 can be used.

```
See also the set gridmajcolour command.
```

5.23.17 key

```
set key [ <position> ... ] [<xoffset>, <yoffset>]
```

The setting '`key`' determines whether a legend is placed on a plot, and if so, where it should be located on the plot. Issuing the command:

```
set key
```

simply causes a legend to be added to the plot in its default position, usually the plot's upper-right corner. The converse action is achieved by:

```
set nokey
```

or:

```
unset key
```

both of which cause a plot to have no legend. A position for the key may also be specified after the `set key` command, for example:

```
set key bottom left
```

Recognised positions are '`top`', '`bottom`', '`left`', '`right`', '`below`', '`outside`', '`xcentre`' and '`ycentre`'. In addition, if none of these quite achieved the desired result, a positional offset may be specified after one of the position keywords above. 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.

5.23.18 keycolumns

```
set keycolumns <value>
```

The '`keycolumns`' settings sets how many columns the legend of a plot should be arranged into. By default, all of the entries in the legends of plots are arranged in a single vertical list. However, for plots with very large number of datasets, it may be preferably to split this list into several columns. The `set keycolumns` command can be followed by any positive integer, for example:

```
set keycolumns 3
```

5.23.19 label

```
set label <label number> '<text>' [<co-ordinate>] <x>,
                                  [<co-ordinate>] <y>
                                  [rotate <angle>]

<co-ordinate> = ( first | second | screen | graph |
                  axis<axisnumber>                  )
```

The `set label` command can be used to place text labels onto a plot. For example:

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

would place the word 'Hello' at plot co-ordinates (0,0), as measured on the x- and y-axes. The tag '`1`' immediately following the '`label`' keyword is an identification number, and allows the label to be removed later with the `unset label` command. By default the position coordinates of the label are measured relative to the first x- and y-axes, but can be specified in a range of coordinate systems. These are specified as follows:

```
set label 1 'Hello' first 0, second 0
```

As can be seen, the name of the desired coordinate system precedes the position value in that coordinate system. Following gnuplot's nomenclature, the coordinate system `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. The syntax `axisn` may also be used, to use the n th x- or y-axis; for example, `axis3`:

```
set label 1 'Hello' axis3 1, axis4 1
```

A rotation angle may optionally be specified after the keyword '`rotate`' to produce text rotated to any arbitrary angle, measured in degrees counter-clockwise. The following example would produce upward-running text:

```
set label 1 'Hello' 1.2, 2.5 rotate 90
```

5.23.20 linestyle

```
set linestyle <style number> <style specifier> ...
```

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
```

5.23.21 linewidth

```
set linewidth <value>
```

Sets the default linewidth, in units of pt, of the lines used to plot datasets onto graphs with the '`lines`' plot style (see the `plot` command for details of plot styles), for example in the following statement:

```
plot sin(x) with lines
```

The linewidths of individual datasets can be set as follows; the `set linewidth` setting only affects plot statements where no linewidth is manually specified:

```
plot sin(x) with lines linewidth 5.0
```

5.23.22 logscale

```
set logscale [<axis> ... ] <base>
```

The '`logscale`' setting causes an axis to be laid out with logarithmically, rather than linearly, spaced intervals. For example, issuing the command:

```
set log
```

would cause all of the axes of a plot to be scaled logarithmically. Alternatively only one, or a selection of axes, can be set to scale logarithmically as follows:

```
set log x1 y2
```

This would cause the first x- and second y-axes to be scaled logarithmically. Linear scaling can be restored to all axes via:

```
set nolog
```

or:

```
unset log
```

and to only one, or a selection of axes, via:

```
set nolog x1 y2
```

or:

```
unset log x1y2
```

Optionally, a base may be specified at the end of the `set logscale` command, to produce axes labelled in logarithms to arbitrary bases.

5.23.23 multiplot

```
set multiplot
```

Issuing the command:

```
set multiplot
```

causes PyXPlot to enter multiplot mode, which allows many graphs to be plotted together and displayed side-by-side. See Section 3.5 for a full discussion of multiplot mode.

5.23.24 mxtics

See `set xtics`.

5.23.25 mytics

See `set xtics`.

5.23.26 noarrow

```
set noarrow [<arrow number>]
```

Issuing the command:

```
set noarrow
```

removes all arrows, as set using the `set arrow` command, from the current plot. Alternatively, individual arrows can be removed using the syntax:

```
set noarrow 2
```

where the tag '`2`' here is the identification number given to the arrow to be removed when it was initially set using the `set arrow` command.

5.23.27 noaxis

```
set noaxis <axis specification>, ...
```

The `set noaxis` command is equivalent to the `unset axis` command. It should be followed by a comma-separated lists of axes, which are to be removed from the current axis configuration.

5.23.28 nobackup

See `backup`.

5.23.29 nodisplay

See `display`.

5.23.30 nogrid

```
set norgrid [<axis> ... ]
```

Issuing the command `set nogrid` removes gridlines from the current plot. On its own, the command removes all gridlines from the plot, but alternatively, those gridlines connected to the ticks of certain axes can selectively be removed. The syntax for doing this is as follows:

```
set nogrid x1 y2
```

5.23.31 nokey

```
set nokey
```

Issuing the command `set nokey` causes plots to be generated with no legend. See the command `set key` for more details.

5.23.32 nolabel

```
set nolabel [<label number> ... ]
```

Issuing the command:

```
set nolabel
```

removes all text labels, as set using the `set label` command, from the current plot. Alternatively, individual labels can be removed using the syntax:

```
set nolabel 2
```

where the tag '`2`' here is the identification number given to the label to be removed when it was initially set using the `set label` command.

5.23.33 nolinestyle

```
set nolinestyle <style number>
```

The `nolinestyle` setting deletes a line style. For example, the command:

```
set nolinestyle 3
```

would delete the third linestyle, if defined. See the command `set linestyle` for more details.

5.23.34 nologscale

```
set nologscale [<axis> ... ]
```

The `logscale` setting causes an axis to be laid out with logarithmically, rather than linearly, spaced intervals. Conversely, the `nologscale` setting is used to restore linear scaling. For example, issuing the command:

```
set nolog
```

would cause all of the axes of a plot to be scaled linearly. Alternatively only one, or a selection of axes, can be set to scale linearly as follows:

```
set nologscale x1 y2
```

This would cause the first x- and second y-axes to be scaled linearly.

5.23.35 nomultiplot

```
set nomultiplot
```

Issuing the command `set nomultiplot` places PyXPlot into single plotting mode. See above for a detailed discussion of PyXPlot's multiplot and single plot modes. Broadly speaking, single plot mode is used to produce single graphs on their own; multiplot mode is used to produce galleries of many plots side-by-side.

5.23.36 notitle

```
set notitle
```

Issuing the command `set notitle` will cause graphs to be produced with no title at the top.

5.23.37 noxtics

```
set no<axis sepcification>tics
```

This command causes graphs to be produced with no tick marks along their x-axes.

5.23.38 noytics

See `set noxtics`.

5.23.39 origin

```
set origin <x>, <y>
```

The '`origin`' setting controls the default location of graphs on a multiplot. For example, the command:

```
set origin 3,5
```

would cause the next graph to be plotted at position $(3, 5)$ centimetres on the multiplot page. The `set origin` command is of little use outside multiplot mode.

5.23.40 output

```
set output '<filename>'
```

The `output` setting controls the name of the file that is produced for non-interactive terminals (`postscript`, `eps`, `jpeg`, `gif` and `png`). For example:

```
set output 'myplot.eps'
```

causes the output to be written to the file '`myplot.eps`'.

5.23.41 palette

```
set palette <colour>, [<colour> ... ]
```

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 is referred if the user issues a command such as:

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

requesting the fifth colour from the palette. By default, this palette contains a range of distinctive colours, however the user can choose to substitute his own list of colours for these using the `set palette` command. It should be followed by a comma-separated list of colour names, for example:

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

If, after issuing this command, the following plot statement were to be executed:

```
plot sin(x), cos(x), tan(x), exp(x)
```

the first function would be plotted in red, the second in green, and the third in blue. Upon reaching the fourth, the palette would cycle back to red.

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

5.23.42 papersize

```
set papersize (size|<x>, <y>)
```

The `papersize` option sets the size of output produced by the postscript terminal. This can take the form of either a recognised papersize name – a list of these is given below – or a height, width pair of values, both measured in millimetres. For example:

```
set papersize a4
set papersize letter
set papersize 200,100
```

A list of recognised papersizes can be found in Figure 3.1.

5.23.43 pointlinewidth

```
set pointlinewidth <value>
```

The '`pointlinewidth`' setting changes the width of the lines that are used to plot datapoints. For instance:

```
set pointlinewidth 20
```

would cause points to be plotted with lines 20 times the default thickness.

Note that '`pointlinewidth`' can be abbreviated as '`plw`'.

5.23.44 pointsize

```
set pointsize <value>
```

The '`pointsize`' setting changes the size at which points are plotted relative to their default size. It should be followed by a single value, the relative size, which can be any positive number. For example:

```
set pointsize 1.5
```

would cause points to be plotted 1.5 times the default size.

5.23.45 preamble

The `premble` setting changes the preamble that is prepended to each item of text rendered using LaTeX. This allows, for example, different packages to be loaded by default and user-defined macros to be set up.

5.23.46 samples

The `samples` setting determines the number of values along the x-axis at which functions are evaluated when they are plotted. For example:

```
set samples 100
```

causes 100 points to be evaluated. Increasing this value will cause functions to be plotted more smoothly, but also more slowly, and the resulting postscript files generated will be correspondingly larger.

When functions are plotted with the `points` plot style, this also affects the number of points plotted.

5.23.47 size

```
set size (<width>|ratio <ratio>|noratio|square)
```

The setting `size` is deprecated; use `set width` instead. It sets the width of the plot in centimetres. However, the command `set size`, when followed by the keyword `ratio`, is still used to set the aspect ratio of plots. See the '`ratio`' setting below for details.

noratio

```
set size noratio
```

Running:

```
set size noratio
```

resets PyXPlot to produce plots with its default aspect ratio, which is the golden section. Other aspect ratios can be set with the `set size ratio` command.

ratio

```
set size ratio <ratio>
```

The command:

```
set size ratio x
```

sets the aspect ratio of plots produced by PyXPlot. The height of resulting plots will equal the plot width, as set by the `set width` command, muliplied by this aspect ratio. The value x in the above statement can be substituted with any positive value, for example:

```
set size ratio 2.0
```

would cause PyXPlot to produce plots that are twice as high as they are wide.

The default aspect ratio which PyXPlot uses is a golden ratio of $2/(1+\sqrt{5})$, which matches that of a sheet of A4 paper.

square

```
set size square
```

The command:

```
set size square
```

sets PyXPlot to produce square plots, i.e. with unit aspect ratio. Other aspect ratios can be set with the `set size ratio` command.

5.23.48 style

```
set style { data | function } <style modifier> ...
```

The `set style data` command affects the default style that data from a file is plotted with. Likewise the `set style function` command changes the default style that functions are plotted with. Any valid style modifier can be used. For example:

```
set style data points
set style function lines linestyle 1
```

would cause datafiles to be plotted by default using points and functions using lines with the first defined linestyle.

5.23.49 terminal

```
set terminal <terminal type> [<option> ... ]
```

Syntax:

```
set terminal { X11_singlewindow | X11_multiwindow | X11_persist |
               postscript | eps | pdf | gif | png | jpg }
             { colour | color | monochrome }
             { portrait | landscape }
             { invert | noinvert }
             { transparent | solid }
             { enlarge | noenlarge }
```

The `set terminal` command controls the graphic format in which PyXPlot should output plots, for example setting whether it should output plots to files or display them in a window on the screen. Various options can also be set within many of the graphic formats which PyXPlot supports using this command.

The following graphic formats are supported: `X11_singlewindow`, `X11_multiwindow`, `X11_persist`, `postscript`, `eps`, `pdf`, `gif`, `jpeg`, `png`. To select one of these formats, simply type the name of the desired format after the `set terminal` command. To obtain more details on each, see the subtopics below.

The following settings, which can also be typed following the `set terminal` command, are used to change the options within some of these graphic formats: `colour`, `monochrome`, `enhanced`, `noenhanced`, `portrait`, `landscape`, `invert`, `noinvert`, `transparent`, `solid`, `enlarge`, `noenlarge`. Details of each of these can be found below.

colour

The `colour` terminal option causes plots to be produced in colour.

color

The `color` terminal option is provided for the convenience of users unable to spell `colour`.

enlarge

The `enlarge` terminal option causes the complete plot to be enlarged or shrunk to fit the current paper size.

eps

```
set terminal eps [<option> ... ]
```

Sends output to eps files. The filename to which output is to be sent should be set using the `set output` command; the default is '`pyxplot.eps`'. This terminal produces encapsulated postscript suitable for including in, for example, LaTeXdocuments.

gif

```
set terminal gif [<option> ... ]
```

The `gif` terminal renders output as gif files. The filename to which output is to be sent should be set using the `set output` command; the default is `pyxplot.gif`. The number of dots per inch used can be changed using the dpi option; the filename using `set output`. Transparent gifs can be produced with the `transparent` option. Also of relevance is the `invert` option for producing gifs with inverted colours.

invert

The `invert` terminal option causes the bitmap terminals (`gif`, `jpeg`, `png`) to produce output with inverted colours. Useful for producing plots for slideshows, where bright colours on a dark background may be desired.

jpeg

```
set terminal jpeg [<option> ... ]
```

The `jpeg` terminal renders output as jpeg files. The filename to which output is to be sent should be set using the `set output` command; the default is `pyxplot.jpg`. The number of dots per inch used can be changed using the dpi option. Of relevance is the `invert` option for producing jpegs with inverted colours.

landscape

The `landscape` terminal option causes PyXPlot's output to be displayed in rotated orientation. Useful for printing as you get more on your sheet of paper that way around; probably less useful for plotting things on screen.

monochrome

The `monochrome` terminal option causes plots to be rendered in black and white; by default, different dash styles are used to differentiate between lines on plots with several datasets.

noenlarge

The `noenlarge` terminal option causes the output not to be scaled (the opposite of `enlarge` above).

noinvert

The `noinvert` terminal option causes the bitmap terminals (`gif`, `jpeg`, `png`) to produce normal output without inverted colours. The converse of `inverse`.

pdf

```
set terminal pdf [<option> ... ]
```

The `pdf` terminal options causes pdf format output files to be produced.

png

```
set terminal png [<option> ... ]
```

The `png` terminal renders output as png files. The filename to which output is to be sent should be set using the `set output` command; the default is `pyxplot.png`. The number of dots per inch used can be changed using the dpi option; the filename using `set output`. Transparent pngs can be produced with the `transparent` option. Also of relevance is the `invert` option for producing pngs with inverted colours.

portrait

The `portrait` terminal option causes PyXPlot's output to be displayed in upright (normal) orientation.

postscript

```
set terminal postscript [<option> ... ]
```

Sends output to postscript files. The filename to which output is to be sent should be set using the `set output` command; the default is `pyxplot.ps`. This terminal produces non-encapsulated postscript suitable for sending directly to a printer.

solid

The solid option causes the `gif` and `png` terminals to produce output with a non-transparent background. The converse of `transparent`.

transparent

The `transparent` terminal option causes the `gif` and `png` terminals to produce output with a transparent background.

X11_multiwindow

Displays plots on the screen (in X11 windows, using ghostview). Each time a new plot is generated it 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.

X11_persist

Displays plots on the screen in X11 windows, using ghostview. Each time a new plot is generated it appears in a new window, and the old plots remain visible. When PyXPlot is exited the windows remain in place until they are closed manually.

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. This terminal is the default when running interactively.

5.23.50 textcolour

```
set textcolour <colour>
```

The '`textcolour`' setting changes the colour of all text displayed on a plot. For example:

```
set textcolour red
```

causes all text labels, including the labels on graph axes and legends, etc. to be rendered in red. Any of the recognised colour names listed in Section 4.6 can be used.

5.23.51 texthalign

```
set texthalign ( left | centre | right )
```

The '`texthalign`' setting controls how text labels, placed on plots using the `set label` command, and upon multiplots using the `text` command, are justified horizontally with respect to their specified positions. Three options are available:

```
set texthalign left
set texthalign centre
set texthalign right
```

5.23.52 textvalign

```
set textvalign ( bottom | center | top )
```

The '`textvalign`' setting controls how text labels, placed on plots using the `set label` command, and upon multiplots using the `text` command, are justified vertically with respect to their specified positions. Three options are available:

```
set textvalign bottom
set textvalign centre
set textvalign top
```

5.23.53 title

```
set title '<title>'
```

The '`title`' setting can be used to set a title for a plot, to be displayed above it. For example, the command:

```
set title 'foo'
```

would cause a title 'foo' to be displayed above a graph. The easiest way to remove a title, having set one, is via:

```
set title ''
```

5.23.54 width

```
set width <value>
```

The `width` setting controls the size of a graph. For example:

```
set width 10
```

sets output to be 10 centimetres in width. For the bitmap terminals (`gif`, `jpg` and `png`) this setting, in conjunction with the `dpi` setting, controls the number of pixels across the final image.

5.23.55 xlabel

```
set xlabel '<text>'
```

The `xlabel` setting controls the label placed on its x-axis (abscissa). For example:

```
set xlabel '$x$'
```

sets the label on the x-axis to 'x'. Labels can be placed on higher axes by inserting their number after the 'x', for example:

```
set x10label 'foo'
```

would label the tenth x axis.

Similarly, labels can be placed on y-axes as follows:

```
set ylabel '$y$'
set y2label 'foo'
```

5.23.56 xrange

```
set x[<axisnumber>]range '<text>'
```

The `xrange` setting controls the range of values along the x-axes of plots. For function plots, this is also the domain across which the function will be evaluated. For example:

```
set xrange [0:10]
```

sets the first x axis to be between 0 and 10. Higher numbered axes may be referred to be inserting their number after the x; y-axes similarly be replacing the x with a y. Hence:

```
set y23range [-5:5]
```

sets the range of the 23rd y-axis to be between -5 and 5. The following command:

```
set xrange [:10]
```

would set the x-axis to have an upper limit of 10, but an automatically-scaling lower-limit.

5.23.57 xticdir

```
set (x|y)[<axisnumber>]ticdir (inward|outward|both)
```

The '`xticdir`' setting can be used to set whether the ticks along the x-axis of a plot point inwards, towards the graph, as by default, or outwards, towards the numeric labels along the axis. They can also be set to point in both directions simultaneously. The syntax for this is as follows:

```
set xticdir inward
set xticdir outward
set xticdir both
```

The same setting can also be made on higher numbered axes, by inserting their numbers after the ‘x’, for example:

```
set x10ticdir outward
```

Similarly, the ‘x’ can be substituted with a ‘y’ to set the directions of ticks on vertical axes:

```
set yticdir inward
set y10ticdir both
```

5.23.58 xtics

```
set [m]x[<axisnumber>]tics
         [axis|border|inward|outward|both]
         [auto
          | [<minimum>,] <increment[, <maximum>]
          | ( '<label>' <position> ... )
         ]
```

The `xtics` option specifies the positions of tick marks on the x-axis (similarly, `ytics` acts on the y-axis). One can specify:

- The axis to modify; if none is specified, then the command acts upon all axes.
- `mxtics` to alter the placement of minor tic marks.
- The keywords `inward`, `outward` and `both`, which alter the directions of the tics. `axis` is an alias for `inward`, `border` for `outward`.
- The `autofreq` keyword restores automatic placement of the tics
- If `minimum`, `increment`, `maximum` are specified, then ticks are placed at evenly spaced intervals between the specified limits. In the case of logarithmic axes, increment is applied multiplicatively.
- The final form allows ticks to be placed on an axis manually with individual labels.

Two examples:

```
set xtics 2 1 5
```

will set tick marks on the x-axis at positions 2, 3, 4 and 5.

```
set x2tics ("a" 2, "b" 3)
```

will set tick marks on the second x-axis at positions 2 and 3 reading ‘a’ and ‘b’ respectively.

5.23.59 ylabel

See `xlabel`.

5.23.60 yrange

See `xrange`.

5.23.61 yticdir

See `xticdir`.

5.23.62 ytics

See `xtics`.

5.24 show

```
show ( all | settings | axes | variables | functions |
       <parameter> ...                                 )
```

The `show` command displays the values of PyXPlot's internal parameters. For example:

```
show pointsize
```

will display the current default point size.

Details of the various settings that can be shown can be found under the `set` command; any keyword which can follow the `set` command can also follow the `show` command.

In addition, `show all` shows the configuration state of all aspects of PyXPlot. The command `show settings` shows all of PyXPlot's settings, as distinct from variables, functions and axes. `show axes` shows the configuration of all of PyXPlot's axes. `show variables` lists all of the currently defined variables. And finally, `show functions` lists all of the current user-defined functions.

5.25 spline

```
spline [<range specification>] <function name> '<filename>'
       [index <index specification>] [every <every specification>]
       [using <using specification>]
```

The `spline` command fits a spline to a datafile. A special function is created that represents the spline fit and can be used in the same way as any other user-defined function. For example:

```
spline f() 'data.1'
```

would create a function $f(x)$ that is a fit to the data in the file `data.1`. By default, the `spline` command uses the first two columns of a data file in a manner analogous to the plot command. The `index`, `every` and `using` modifiers can be used in the same way as in the `plot` command to select which parts of the datafile should be used; see the `datafile` section for more details.

Note that trying to generate splines of multi-valued functions will not, in general, produce useful results.

5.26 text

```
text '<text string>' [at <x>, <y>] [rotate <angle>]
```

The `text` command is primarily part of the multiplot environment; it can be used to add blocks of text to a multiplot. It can, however, also be used in single plot mode, in a way that is described below. As always in PyXPlot, the text is rendered using LaTeX. An example would be:

```
text 'Hello world' at 0,2
```

which would render the text 'Hello world' at position $(0, 2)$, measured in centimetres. The alignment of the text item with respect to this position can be set using the `set texthalign` and `set textvalign` commands.

A rotation angle may optionally be specified after the keyword '`rotate`' to produce text rotated to any arbitrary angle, measured in degrees counter-clockwise. The following example would produce upward-running text:

```
text 'Hello' at 1.5, 3.6 rotate 90
```

Outside of multiplot mode, the text command can be used to produce images consisting simply of one single text item. This can be useful for importing LaTeXed equations as gif images into slideshow programs such as Microsoft Powerpoint which are incapable of producing such neat mathematical notation by themselves.

5.27 undelete

```
undelete <item number>, ...
```

The `undelete` command is part of the multiplot environment; it can be used to reverse the effect of deleting a multiplot item with the `delete` command. The desired item to be undeleted should be identified using the reference number which it was given when it was created; it would have been displayed on the terminal at that time. For example:

```
undelete 1
```

will cause the previously item numbered `1` to reappear.

5.28 unset

```
unset <setting>
```

The `unset` command causes a setting that has been changed using the set command to be returned to its default value. For example:

```
unset linewidth
```

returns the linewidth to its default value.

The list of keywords which can follow the `unset` command are essentially the same as those which can follow the `set` command.

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:

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

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 $\mathrm{P}\left(\mathbf{u}|\left\{\mathbf{x}_i, f_i, \sigma_i\right\}\right)$ is equivalent to maximising $\mathrm{P}\left(\left\{f_i\right\}|\mathbf{u}, \left\{\mathbf{x}_i, \sigma_i\right\}\right)$.

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:

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

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 $\mathrm{P}\left(\mathbf{u}|\left\{\mathbf{x}_i, f_i, \sigma_i\right\}\right)$ 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 $\mathrm{P}\left(\mathbf{u}|\left\{\mathbf{x}_i, f_i, \sigma_i\right\}\right)$ 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:

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

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 $\mathrm{P}\left(\mathbf{u} | \{\mathbf{x}_i, f_i, \sigma_i\}\right)$ about $\mathbf{u}^0$.[1] We may write the sum of these terms, which we denote Q, in matrix form:

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

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:

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

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:

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

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:

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

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:

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

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}:

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

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:

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

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

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

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:

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

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:

$$C_{ij} = \frac{V_{ij}}{\sigma_i \sigma_j} \qquad (6.16)$$

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 $\mathrm{P}\left(\sigma_{\mathrm{data}} | \left\{\mathbf{x}_i, f_i\right\}\right)$. Returning once again to Bayes' Theorem, we can write:

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

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

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

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 $\mathrm{P}\left(\left\{f_i\right\} | \sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}, \left\{\mathbf{u}_i\right\}\right)$ 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:

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

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:

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

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

Chapter 7

ChangeLog

2007 Feb 26: PyXPlot 0.6.3

Summary:

Second PyXPlot beta-release. The most significant change is the introduction of a new command-line parser, with greatly improved handling of complex expressions and much more meaningful syntax error messages. Multi-platform compatibility has also been massively improved, and dependencies loosened. A small number of new commands have been added; most notable among them are the `jpeg` and `eps` commands, which embed images in multiplots.

Details – New and Extended Commands:

- `jpeg`
- `eps`
- `set xtics` and `set mxtics`
- `text` and `set label` commands extended to allow text rotation.
- `set log` command extended to allow the use of logarithms with bases other than 10.
- `set preamble`
- `set term enlarge | noenlarge`
- `set term pdf`
- `set term x11_persist`

Details – Eased System Requirements:

- Requirement on Python 2.4 minimum eased to version 2.3 minimum.
- Requirements on scipy and readline eased; PyXPlot will now work in reduced form when they are absent, though they are still strongly recommended.
- dvips and ghostscript are no longer required.

Details – Removed Commands:

Due to a general refinement of PyXPlot's API, some of the less sensible pieces of syntax from Version 0.5 are no longer supported. The author apologises for any inconvenience caused.

- The `delete_arrow`, `delete_text`, `move_text`, `undelete_arrow` and `undelete_text` commands have been removed from the PyXPlot API. The `move`, `delete` and `undelete` commands should now be used to act upon all types of multiplot objects.
- The `set terminal` command no longer accepts the `enhanced` and `noenhanced` modifiers. The `postscript` and `eps` terminals should be used instead.
- The `select` modifier, used after the `plot`, `replot fit` and `spline` command can now only be used once; to specify multiple `select` criteria, use the `and` logical operator.

2006 Sep 09: PyXPlot 0.5.8

First beta-release.

Index