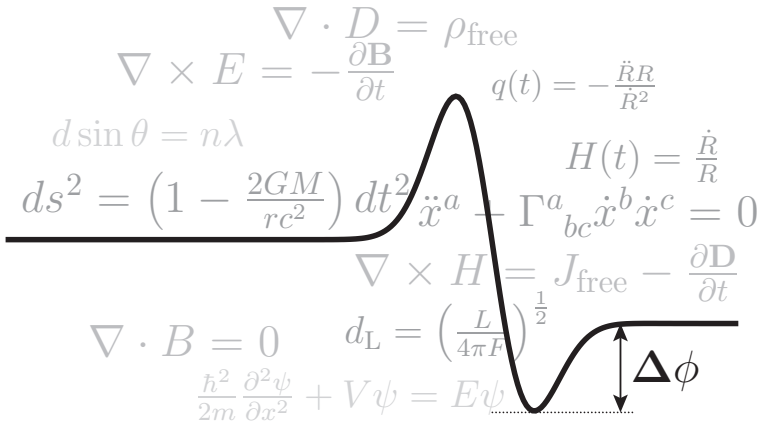
Pyxplot Users' Guide

A Scientific Scripting Language, Graph Plotting Suite and Vector Graphics Toolkit.

Version 0.9.2



Lead Developer: Dominic Ford

Lead Tester: Ross Church

Email: `coders@pyxplot.org.uk`

This manual is also available in HTML, at
http://www.pyxplot.org.uk/0.9/doc/html/

September 2012

Contents

List of Figures

List of Examples

Part I

Introduction to Pyxplot

Chapter 1

Introduction

1.1 What is Pyxplot?

Pyxplot is a multi-purpose graph plotting tool, scientific scripting language, vector graphics suite, and data processing package. Its interface is designed to make common tasks – e.g., plotting labelled graphs of data – accessible via short, simple, intuitive commands.

But these commands also take many optional settings, allowing their output to be fine-tuned into styles appropriate for printed publications, talks or websites. Pyxplot is simple enough to be used without prior programming experience, but powerful enough that programmers can extensively configure and script it.

A scientific scripting language

Pyxplot doesn't just plot graphs. It's a scripting language in which variables can have physical units. Calculations automatically return results in an appropriate unit, whether that be kilograms, joules or lightyears. Data files can be converted straightforwardly from one set of units to another. Meanwhile Pyxplot has all the other features of a scripting language: flow control and branching, string manipulation, complex data types, an object-oriented class structure and straightforward file I/O. It also supports vector and matrix algebra, can integrate or differentiate expressions, and can numerically solve systems of equations.

A vector graphics suite

The graphical canvas isn't just for plotting graphs on. Circles, polygons and ellipses can be drawn to build vector graphics. Colors are a native object type for easy customisation. For the mathematically minded, Pyxplot's canvas interfaces cleanly with its vector math environment, so that geometric construction is easy.

A data processing package

Pyxplot can interpolate data, find best-fit lines, and compile histograms. It can Fourier transform data, calculate statistics, and output results to new data files. Where fine control is needed, custom code can be used to process every data point in a file.

1.2 Compatibility with gnuplot

Pyxplot's plotting interface is very similar to that of gnuplot: the commands used for plotting simple graphs in the two programs are virtually identical. Gnuplot users will have a head start with Pyxplot – simple gnuplot scripts often work in Pyxplot with minimal modification – although the syntax used for advanced plotting tasks often differs. Although Pyxplot's programming and mathematical environment is hugely extended over gnuplot's to provide a scripting language and vector graphics environment, we have followed the latter's preference for short simple command line syntax.

1.3 The structure of this manual

This manual serves both as a tutorial guide to Pyxplot, and also as a reference manual. Part I provides a step-by-step tutorial and overview of Pyxplot's features, including numerous worked examples. Part II provides a detailed survey of Pyxplot's plotting and vector graphics commands. Part III provides an alphabetical reference to all of Pyxplot's commands, mathematical functions and plotting options. Finally, the appendices provide information which is likely to be of more specialist interest.

1.4 An introductory tour

This section provides an overview of the wide range of tasks for which Pyxplot can be used. Detailed explanations of the syntax of Pyxplot commands will follow in later chapters, but most of the examples here will work if entered directly at a Pyxplot command prompt.

The mathematical environment

Pyxplot's mathematical environment comes with many standard functions built-in. To see a list of them[1], type

```
show functions
```

The `show` command is Pyxplot's interactive documentation system; to obtain a list of things that can be shown type

```
show
```

Pyxplot is an object-orientated language, and its built-in functions live in a module called `defaults`. Its members may be listed by `print`ing the module object.

```
print defaults
```

Taking as an example the built-in function `log10(x)`, it can be evaluated as in almost any other programming language. The `defaults` module is special in that its functions are always accessible to the user's namespace:

[1] See also Chapter 12.

```
pyxplot> print log10(5)
0.69897
```

This function returns numerical data, but Pyxplot has other data types too. For example, the `primeFactors` function returns a list:

```
pyxplot> print primeFactors(1001)
[7, 11, 13]
```

The `rgb(r,g,b)` function returns a color object, which can be used in Pyxplot's vector graphics commands:

```
pyxplot> print rgb(1,1,0)
rgb(1,1,0)
```

and the `time.fromUnix(t)` function returns a date object from a Unix time:

```
pyxplot> print time.fromUnix(946684800)
Sat 2000 Jan 1 00:00:00 UTC
```

Many commonly-used physical constants are built into Pyxplot's physics module, `phy`. For example, the speed of light:

```
pyxplot> print phy.c
299792.46 km/s
```

Numbers in Pyxplot have **physical units**, and hence the speed of light is displayed in km/s. If you would rather know how many miles light travels in a year, you can change the display unit, here making use of the fact that the variable `ans` is always set to the result of the last calculation:

```
pyxplot> print phy.c
299792.46 km/s
pyxplot> set unit preferred miles/year
pyxplot> print ans
5.87849967e+12 mi/yr
```

It is easy to use Pyxplot as a **desktop calculator** to solve many problems which would conventionally need careful conversion between physical units. For example:

- What is 80°F in Celsius?

  ```
  pyxplot> print 80*unit(oF) / unit(oC)
  26.666667
  ```

- How long does it take for light to travel from the Sun to the Earth?[2]

  ```
  pyxplot> print unit(AU) / phy.c
  499.00478 s
  ```

[2]The astronomical unit (AU) is a unit used by astronomers, equal to the average distance of the Earth from the Sun.

- What wavelength of light corresponds to the ionisation energy of hydrogen (13.6 eV)?[3]

```
pyxplot> print phy.c * phy.h / (13.6 * unit(eV))
91.164844 nm
```

- What is the escape velocity of the Earth?[4]

```
pyxplot> print sqrt(2 * phy.G * unit(Mearth) / unit(Rearth))
11.186605 km/s
```

Graph plotting

The simplest way to plot a graph in Pyxplot is simply to follow the `plot` command with the name of a function to be plotted, e.g.:

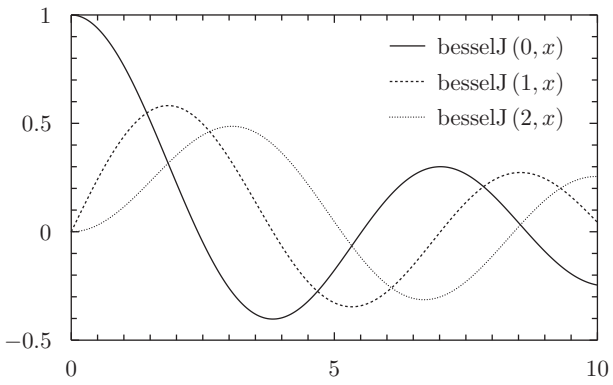
```
plot sin(x)
```

If a data file is to be plotted, its filename is put in place of a named function. In this example the fifth column of a data file is plotted against the second, including only those data points where the fourth column is larger than two:

```
plot 'data.dat' using 2:5 select $4>2
```

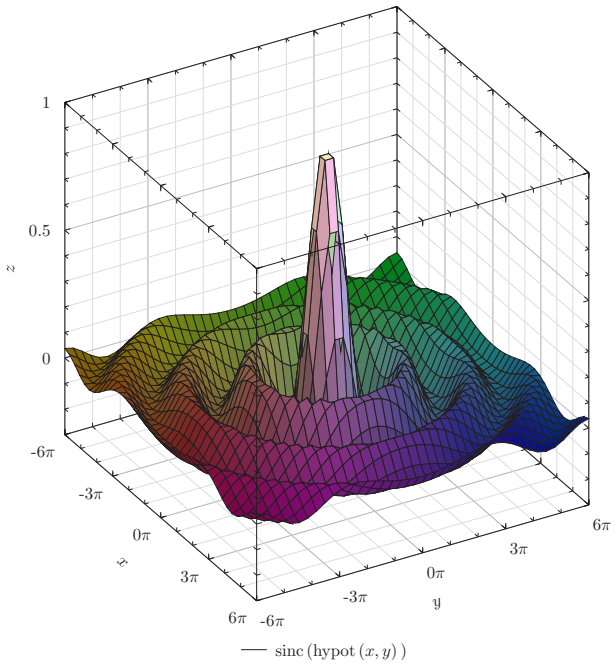
In the example below, three Bessel functions are plotted over specified horizontal and vertical ranges:

```
plot [0:10][-0.5:1] besselJ(0,x), besselJ(1,x), besselJ(2,x)
```

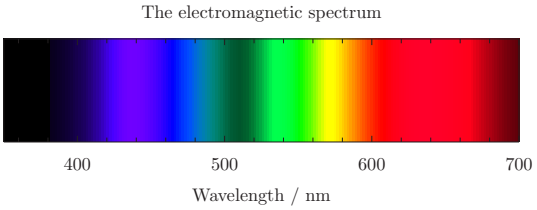


With a little additional configuration, it is possible to produce three-dimensional plots like this (this example is taken from Section 8.14.1; see Example 27):

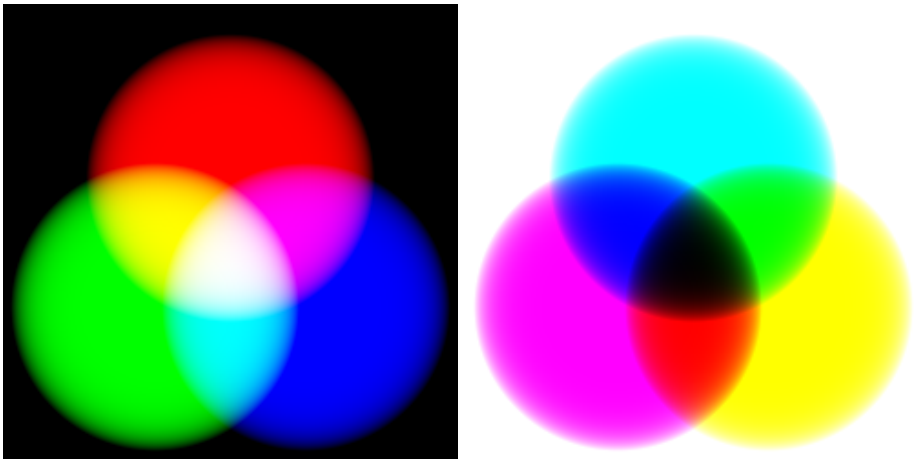
[3] The electron volt (eV) is a unit of energy used by physicists.
[4] The Earth radius and Earth mass are defined as units in Pyxplot.



It is also possible to produce colormaps with custom color scales; this is documented in full in Section 8.12. Pyxplot includes functions for converting wavelengths of light into colors; they are used here to create a color map of the electromagnetic spectrum:



This pair of images demonstrates RGB color mixing (see Example 8.12.1):



Generating data tables

Pyxplot can output tables of data to disk, using a similar syntax to that used for plotting graphs. The data can either be sampled from functions, or read in from another data file:

```
tabulate tan(x)
```

A common application of the `tabulate` command is filter or re-format the contents of data files. For example, the command below takes only the third and seventh columns out of a data file, and converts the latter from degrees into radians:

```
tabulate 'data.dat' using 3:$7*unit(deg)/unit(rad)
```

The same effect could be achieved by setting radians as the default unit of angle:

```
set unit of angle radians
tabulate 'data.dat' using 3:$7*unit(deg)
```

More sophisticated data processing is also possible; this example produces a histogram of the values in the fourth column of a datafile, and then outputs that histogram as a new data file:

```
histogram h() 'data.dat' using 4
tabulate h(x)
```

Solving equations

Pyxplot can solve systems of equations numerically; the following example evaluates $\int_{0\,\mathrm{s}}^{2\,\mathrm{s}} x^2\,\mathrm{d}x$:

```
pyxplot> print int_dx(x**2,0*unit(s),2*unit(s))
2.6666667 s**3
```

The `solve` command can be used to solve systems of simultaneous equations, such as this system with two variables:

```
pyxplot> solve x+y=1 , 2*x+3*y=7 via x,y
pyxplot> print "x=%s; y=%s"%(x,y)
x=-4; y=5
```

The `minimise` and `maximise` commands find the extrema of functions; here they are used to find the minimum of the function $\cos(x)$ closest to $x = 0.5$:

```
pyxplot> x=0.5
pyxplot> minimise cos(x) via x
pyxplot> print x
3.1415927
```

All of the examples shown so far have used only real numbers, but Pyxplot can also perform algebra on complex numbers. By default, evaluation of

`sqrt(-1)` throws an error – the emergence of complex numbers is often an indication that a calculation has gone wrong – but complex arithmetic can be enabled by typing

```
pyxplot> set numerics complex
pyxplot> print sqrt(-1)
i
```

Many of the mathematical functions which are built into Pyxplot can take complex arguments, for example

```
pyxplot> set numerics complex
pyxplot> print exp(2+3*i)
(-7.3151101+1.0427437i)
pyxplot> print sin(i)
1.1752012i
pyxplot> print arg(2+3*i)
0.98279372
pyxplot> print Re(2+3*i)
2
```

Vector graphics

Pyxplot's graph-plotting canvas can also be used for drawing general vector graphics, using simple commands such as:

```
box from -8,-4 to 8,4 with fillcolor green
text "Pyxplot" at 2,3
arrow from 0,0 to -4,2
line from -5,0 to 5,0
```

These commands are described in detail in Chapter 10. They interface neatly to the vector data type in Pyxplot's mathematical environment, to ease geometric construction. Thus, it is quite possible for mathematically-minded users to multiply transformation matrices with position vectors on the graphics canvas to calculate where objects should be drawn. The following example uses a rotation matrix to draw a big arrow at angle θ to the vertical:

```
rotate(a) = matrix( [[cos(a),-sin(a)], \
                     [sin(a), cos(a)] ] )
pos = vector(0,5)*unit(cm)
theta = 30*unit(deg)
arrow from 0,0 to rotate(theta)*pos with linewidth 3
```

1.5 License

the web, from the following location:
http://www.gnu.org/copyleft/gpl.html.

1.6 Spelling conventions

Throughout this manual, US English is used. However, where spelling differs between US and UK English, Pyxplot recognises both variants. For example, where the word `color` appears in Pyxplot syntax, it may also be spelt `colour`; `minimize` may also be spelt `minimise`; `gray` may also be spelt `grey`; `neighbor` may also be spelt `neighbour`; etc.

1.7 Acknowledgments

Pyxplot builds on ideas from several pre-existing open-source software projects. We like gnuplot's simple and intuitive interface, and Pyxplot's command syntax is intentionally very similar, to the point of backwards compatibility in many cases. Even when designing the entirely new parts of Pyxplot's syntax, we have followed gnuplot's preference for short simple command-line syntax. Early versions of Pyxplot utilised the PyX graphics library for Python, and we have borrowed many ideas from it in our new output engine.

Several people have contributed code to Pyxplot. Michael Rutter provided us with a copy of his public domain code for converting bitmap images into PostScript, which we used in the implementation of the `image` command and the `colormap` plot style. Matthew Smith provided C implementations of the Airy functions and the Riemann zeta function for general complex inputs, and helped test Pyxplot's mathematical environment. Zoltán Vörös worked on our development team from 2010 until 2011. John Walker has published public domain code implementing RGB rendering of the electromagnetic spectrum, which we use in the `colors.wavelength()` function.

We would also like to thank all the users who have got in touch with us by email since Pyxplot was first released on the web in 2006. Your feedback and suggestions have been gratefully received.

Final thanks go to our team of alpha testers, without whose work Pyxplot would doubtless still contain many more bugs. Especial thanks go to Rachel Holdforth and Stuart Prescott.

Chapter 2

Installation

In this chapter we describe how to install Pyxplot on a range of UNIX-like operating systems.

Pyxplot works on most UNIX-like operating systems. We have tested it under Linux, Solaris and MacOS, and believe that it should work on other similar POSIX systems. We regret that it is not available for Microsoft Windows, and have no plans for porting it at this time.

2.1 Installation within Linux distributions

By far the easiest way to install Pyxplot under Linux is to use your distribution's package manager. If you use a recent release of Gentoo[1], Ubuntu[2] or Debian[3], your package manager can install Pyxplot and all its dependencies for you, though the packaged version may be several months behind the latest release. Please note that this manual describes Pyxplot 0.9.x, which is a very substantial upgrade to version 0.8.x. To install the packaged version of Pyxplot under Debian or Ubuntu, simply type:

```
apt-get install pyxplot gv
```

Users of other distributions, or who want a newer version of Pyxplot, should use the `.tar.gz` archives available from the Pyxplot website. The process is described below.

2.2 System requirements

Pyxplot depends on the following software packages, which are not included in the source tarball:

- fftw (version 2 or, preferably, 3+)
- gcc and make

[1] See http://gentoo-portage.com/sci-visualization/pyxplot
[2] See http://packages.ubuntu.com/pyxplot
[3] See http://packages.debian.org/pyxplot

- Ghostscript
- The Gnu Scientific Library (version 1.10+)
- ImageMagick
- latex (version 2ϵ; a full installation is likely to be required in distributions which offer a choice)
- libpng (version 1.2+)
- libxml2 (version 2.6+)
- zlib

It is very strongly recommended that the following software packages also be installed:

- cfitsio – required for Pyxplot to be able to plot data files in FITS format.
- Ghostview (or `ggv`) – required for Pyxplot to be able to display plots live on the screen; Pyxplot remains able to generate image files on disk without it. Alternatively, the `set viewer` command within Pyxplot allows a different PostScript viewer to be used.
- gunzip – required for Pyxplot to be able to plot compressed data files in `.gz` format.
- The Gnu Readline Library (version 5+) – required for Pyxplot to be able to provide tab completion and command histories in Pyxplot's interactive command-line interface.
- libkpathsea – required to efficiently find the fonts used by latex.
- wget – required for Pyxplot to be able to plot data files directly from the Internet.

In the case of the recommended packages, Pyxplot tests for the availability of each when it is installed and issues a warning if any are not found. Installation can proceed, but some of Pyxplot's features will be disabled. If they are later added to the system, Pyxplot should be reinstalled to take advantage of their presence.

2.2.1 Dependencies in Debian and Ubuntu

Debian and Ubuntu users can find the above software in the following packages[4]:

[4]The package names listed here are correct as of Debian Squeeze and Ubuntu 12.04 (Precise). However, packages occasionally change name between versions.

`fftw3-dev`, `gcc`, `ghostscript`, `gv`, `imagemagick`, `libc6-dev`, `libcfitsio3-dev`, `libgsl0-dev`, `libkpathsea-dev`, `libpng12-dev`, `libreadline-dev`, `libxml2-dev`, `make`, `texlive-latex-extra`, `texlive-latex-recommended`, `texlive-fonts-extra`, `texlive-fonts-recommended`, `wget`, `zlib1g-dev`.

These packages may be installed from a command prompt by typing, all on one line:

```
sudo apt-get install fftw3-dev gcc ghostscript gv imagemagick
       libc6-dev libcfitsio3-dev libgsl0-dev libkpathsea-dev
       libpng12-dev libreadline-dev libxml2-dev make
       texlive-latex-extra texlive-latex-recommended
       texlive-fonts-extra texlive-fonts-recommended wget
       zlib1g-dev
```

2.2.2 Dependencies in MacOS

Users of MacOS X can find the above software in the following MacPorts packages:

`cfitsio`, `fftw-3`, `ghostscript`, `gsl-devel`, `gv`, `ImageMagick`, `libpng`, `libxml2`, `readline-5`, `texlive`, `wget`, `zlib`.

It may then be necessary to run the command

```
export C_INCLUDE_PATH=/opt/local/include
```

before running the `configure` script below.

2.3 Installation from source archive

First, download the required archive can be downloaded from the front page of Pyxplot website – http://www.pyxplot.org.uk. It is assumed that the packages listed above have already been installed; if they are not, you will need to either install them yourself, if you have superuser access to your machine, or contact your system administrator.

- Unpack the distributed .tar.gz:

  ```
  tar xvfz pyxplot_0.9.2.tar.gz
  cd pyxplot-0.9.2
  ```

- Run the installation script:

  ```
  ./configure
  make
  ```

- Finally, start Pyxplot:

  ```
  ./bin/pyxplot
  ```

2.3.1 System-wide installation

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

```
sudo 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 user of the system, simply by typing:

```
pyxplot
```

Chapter 3

First steps with Pyxplot

This chapter provides an overview of the commands used to produce plots in Pyxplot. The commands it covers have very similar syntax to gnuplot; if you have used gnuplot in the past, you can find an approximate list of differences between Pyxplot and gnuplot in Appendix B.

The introduction to plotting provided by this chapter will be extended in Chapter 8, which is a complete guide to Pyxplot's plot styles.

3.1 Getting started

The simplest way to start Pyxplot is to type `pyxplot` at a shell prompt. This starts an interactive session, and produces a Pyxplot command-line prompt into which commands can be typed. Pyxplot can be exited either by typing `exit`, `quit`, or by pressing CTRL-D. Various switches can be specified on the shell command line to modify Pyxplot's behaviour; these are listed in Box 1. Of particular interest may be the switches `-c` and `-m`, which change between the use of color-highlighted (default) and non-colored text.

Typing commands into interactive terminals is likely to be a sufficient way for a beginner to drive Pyxplot, but as tasks grow more complicated, more commands are needed to set up plots. It is likely to become preferable to store these commands in text files called *scripts*. Once such a script has been written, it can be executed automatically by passing the filename of the command script to Pyxplot on the shell command line, for example:

```
pyxplot foo.ppl
```

In this case, Pyxplot executes the commands in the file `foo.ppl` and then exits. By convention, we affix the suffix `.ppl` to the filenames of all Pyxplot command scripts. This is not strictly necessary, but it allows Pyxplot scripts to be easily distinguished from other text files in a filing system. The filenames of several command scripts may be passed to Pyxplot on a single command line, indicating that they should be executed in sequence, as in the example:

```
pyxplot foo1.ppl foo2.ppl foo3.ppl
```

From the shell command line, 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 --color`	Use color highlighting[a], as is the default behaviour, to display output in green, warning messages in amber, and error messages in red.[b] These colors can be changed in the `terminal` section of the configuration file; see Section 19.3.4 for more details.
`-m --monochrome`	Do not use color highlighting.

[a] This will only function on terminals which support color output.
[b] The authors apologise to those members of the population who are red/green color blind, but draw their attention to the following sentence.

Box 1: A list of the command line options accepted by Pyxplot.

It is also possible to have a single Pyxplot session alternate between running command scripts autonomously and allowing the user to enter commands interactively. There are two ways of doing this. Pyxplot can be passed the magic filename `--` on the command line, as in the example

```
pyxplot foo1.ppl -- foo2.ppl
```

where the `--` represents an interactive session which commences after the execution of `foo1.ppl` and should be terminated by the user in the usual way, using either the `exit` or `quit` commands. After the interactive session is finished, Pyxplot will automatically execute the command script `foo2.ppl`.

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

```
pyxplot> load 'foo.ppl'
```

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

The `save` command may assist in producing Pyxplot command scripts: it stores to a text file a history of the commands which have been typed into the present interactive session.

3.2 First plots

The core graph-plotting command of Pyxplot is the `plot` command. The following simple example plots the trigonometric function $\sin(x)$:

```
plot sin(x)
```

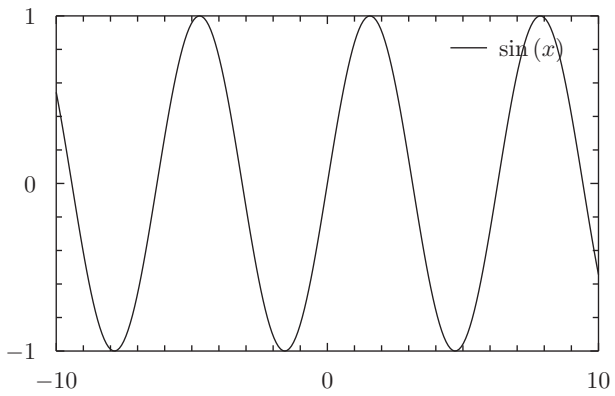
When Pyxplot is used interactively, its command-line environment is based upon the GNU Readline Library. This means that the up- and down-arrow keys can be used to repeat or modify previously executed commands. Each user's command history is stored in his homespace in a history file called `.pyxplot_history`; this file is used by Pyxplot to remember command histories between sessions. Pyxplot's `save` command allows the user to save to a text file a list of the commands which have been typed into the present session, as in the following example:

```
save 'output_filename.ppl'
```

The related `history` command displays on the terminal a history of all of the commands which have been typed into this and previous interactive sessions. The total history can stretch to several hundred lines long, in which case it can be useful to follow the `history` command by an optional number, whereupon it only displays the last n commands, e.g.:

```
history 20
```

Box 2: The storage of command histories in Pyxplot.



This is one of a large number of standard mathematical functions which are built into Pyxplot; a complete alphabetical list of them can be found in Chapter 12.

It is also possible to plot data stored in files. The following would plot data from a file `data.dat`, taking the x-coordinate of each point from the first column of the data file, and the y-coordinate from the second. The data file is assumed to be in plain text format[1], with columns separated by whitespace and/or commas[2]:

```
plot 'data.dat'
```

Several items can be plotted on the same graph by separating them by commas, as in

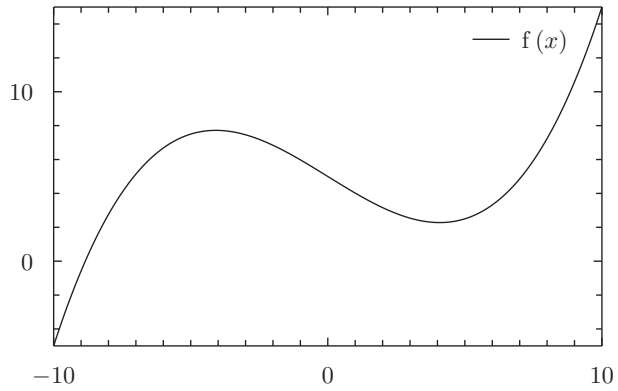
[1] If the filename of a data file ends with a `.gz` suffix, it is assuming to be gzipped plaintext, and is decoded accordingly. Other formats of data file can be opened with the use of input filters; see Section 5.1.

[2] This format is compatible with the Comma Separated Values (CSV) format produced by many applications such as Microsoft Excel.

```
plot 'data.dat', sin(x), cos(x)
```

and it is possible to define one's own variables and functions, and then plot them, as in the example

```
a = 0.02
b = -1
c = 5
f(x) = a*(x**3) + b*x + c
plot f(x)
```



Pyxplot supports almost all of the same mathematical operators as the C programming language; a complete list of them can be found in Table 3.1. If you have experience of similar tricks in C, it is quite possible to write the following expressions in Pyxplot (but don't worry if this is a little over your head):

```
pyxplot> print (a=3)+(b=2)
5
pyxplot> print a>0?"yes":"no"
yes
pyxplot> print "%s  %s"%(++a,b++)
4  2
pyxplot> print (a+=10 , b+=10 , a+b)
27
```

In the final example, the comma operator is used as in C, to return only the value of the final comma-separated expression.

3.3 Comments

As in any programming language, it is good practice to include comments in your code, to help other people (including yourself!) to work out what's going on. Comment lines in Pyxplot scripts should begin with a hash character, as in the example

```
# This is a comment
```

Comments may also be placed on the same line as commands, as in the example

Symbol	Description	Operator Associativity
`**`	Algebraic exponentiation	right-to-left
`-`	Unary minus sign	right-to-left
`--`	Unary decrement	right-to-left
`+`	Unary plus sign	right-to-left
`++`	Unary increment	right-to-left
`not !`	Logical not	right-to-left
∼	Unary one's complement	right-to-left
`*`	Algebraic multiplication	left-to-right
`/`	Algebraic division	left-to-right
`%`	Modulo operator	left-to-right
`+`	Algebraic sum	left-to-right
`-`	Algebraic subtraction	left-to-right
`<<`	Left binary shift	left-to-right
`>>`	Right binary shift	left-to-right
`<`	Magnitude comparison	right-to-left
`>`	Magnitude comparison	right-to-left
`<=`	Magnitude comparison	right-to-left
`>=`	Magnitude comparison	right-to-left
`== <>`	Equality comparison	right-to-left
`!=`	Equality comparison	right-to-left
`&`	Binary and	left-to-right
`^`	Binary exclusive or	left-to-right
`|`	Binary or	left-to-right
`and &&`	Logical and	left-to-right
`or ||`	Logical or	left-to-right
`?:`	Ternary conditional	right-to-left
`=`	Assignment operator	right-to-left
`+= -= *=` `/= %= &=` `^= |=` `<<= >>=`	Assignment operators	right-to-left
`,`	Comma separator	left-to-right

Table 3.1: A list of mathematical operators which Pyxplot recognises, in order of descending precedence. Items separated by horizontal rules are of differing precedence; those not separated by horizontal rules are of equal precedence. The third column indicates whether strings of operators are evaluated from left to right, or from right to left. For example, the expression `x**y**z` is evaluated as `(x**(y**z))`.

```
set nokey # I'll have no key on _my_ plot
```

In both cases, all of the characters following the hash character are ignored.

3.4 Splitting long commands

Long commands may be split over multiple lines, provided that each line of the command is terminated with a backslash character, whereupon the following line will be appended to it. For example:

```
pyxplot> print  2 \
.......>       +3
5
```

Such lines splits are often used in this manual where command lines are longer than the width of the page.

3.5 Printing text

Pyxplot's `print` command can be used to display strings and the results of calculations, as in the following examples:

```
pyxplot> a=2
pyxplot> print "Why don't fish get lost in the ocean?"
Why don't fish get lost in the ocean?
pyxplot> print a
2
```

Multiple items can be displayed one-after-another on a single line by separating them with commas. The following example displays the values of the variable `a` and the function `f(a)` in the middle of a text string:

```
pyxplot> f(x) = x**2
pyxplot> a=3
pyxplot> print "The value of ",a," squared is ",f(a),"."
The value of 3 squared is 9.
```

Strings can be enclosed either in single (') or double (") quotes. Strings may also be enclosed by three quote characters in a row: either ''' or """. Special care needs to be taken when using apostrophes or quotes in single-quote delimited strings, as these characters may be misinterpreted as string delimiters, as in the example:

✗
```
'Robert's data'
```

This easiest way to avoid such problems is to use three quotes:

✔
```
'''Robert's data'''
```

Alternatively, the \ character may be used to escape quote characters. Two backslashes characters – \\ – produce a literal backslash:

Escape sequence	Description
\?	Question mark
\'	Apostrophe
\"	Double quote
\\	Literal backslash
\a	Bell character
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Table 3.2: A complete list of Pyxplot's string escape sequences. These are a subset of those available in C.

✔
```
'Robert\'s data'
"I typed \\' to get an apostrophe"
```

Special characters such as tabs and newlines can be inserted into strings using escape codes, but these are disabled by default. This is because latex uses the backslash as its own escape character, and strings in Pyxplot are commonly used to contain latex commands. So, by default the only escape characters that Pyxplot expands are \', \" and \\.

The use of other escape characters may be enabled by prefixing a string with the character `e`, as in the example `e'\t'` for a tab character. See Table 3.2 for a complete list of escape codes available. For example, the following string is split over three lines:

```
pyxplot> print e'the moon,\nthe moon,\nThey danced by the light of the moon.'
the moon,
the moon,
They danced by the light of the moon.
```

Alternatively, strings may be prefixed with the character `r` to turn off all escape codes, including for quote characters:

```
pyxplot> print r'''I escaped the quote by typing \'.'''
I escaped the quote by typing \'.
```

When many items are being printed together on a line, they can be concatenated together using the `+` operator as above, but it is usually neater to use the string substitution operator, `%`. The operator is preceded by a format string, in which the places where numbers and strings are to be substituted are marked by tokens such as `%e` and `%s`.

The substitution operator is followed by a ()-bracketed list of the quantities which are to be substituted into the format string. This behaviour is similar to that of python's % operator, and of the `printf` command in C, as the following examples demonstrate:

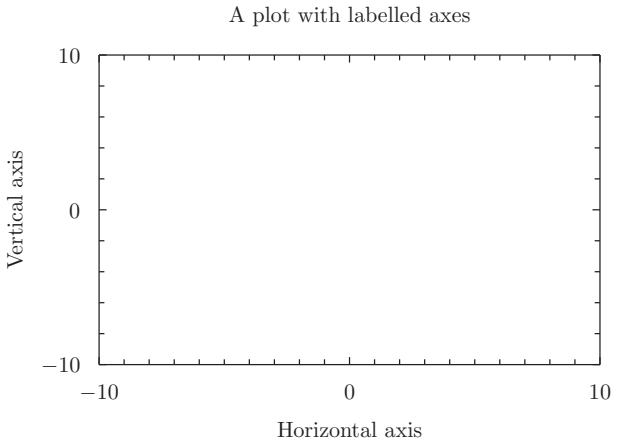
```
pyxplot> f(x)=x**2 ; a=3
pyxplot> print "The value of %d squared is %d."%(a,f(a))
The value of 3 squared is 9.
pyxplot> print "The %s of f(%f) is %d."%("value",sqrt(2), \
.......>                                  f(sqrt(2))     )
The value of f(1.414214) is 2.
```

The detailed behaviour of the string substitution operator, and a full list of the substitution tokens which it accepts, are given in Section 6.2.1.

3.6 Axis labels and titles

Labels can be added to the axes of a plot, and a title put at the top. As with any other strings (see the previous section), labels should be enclosed in either single (') or double (") quotes, as in the following example script:

```
set xlabel "Horizontal axis"
set ylabel "Vertical axis"
set title 'A plot with labelled axes'
plot
```



These labels and title – in fact, all text labels which are ever produced by Pyxplot – are rendered using the latex typesetting system, and so any latex commands can be used to produce custom formatting. This allows great flexibility, but means that care needs to be taken to escape any of latex's reserved characters – i.e. \ & % # { } $ _ ^ or ∼.

Two built-in functions provide some assistance in generating latex labels. The `texify()` function takes as its argument a string containing a mathematical expression, and returns a latex representation of it. The `texifyText()` function takes as its argument a text string, and returns a latex representation of it, with any necessary escape characters added. For example:

```
pyxplot> print texify("sqrt(x**2+1)")
$\displaystyle \sqrt{x\,*\kern-1.5pt *\,\,2+1}\mathrm{}$
pyxplot> a=50
pyxplot> print texifyText("A %d%% increase"%a)
```

```
A 50\% increase
pyxplot> set ylabel texify("cos(x**2)")
```

Special care needs to be taken when typesetting latex expressions that contain apostrophe or quote characters, as these are the string delimiters used by Pyxplot. For ease, it is recommended that latex expressions be enclosed in triple quotes:

✔ `set xlabel """\textrm{J\"org's data}"""`

The reason for recommending this syntax is demonstrated by the examples below, all of which will fail. In

✗ `set xlabel 'My plot's X axis'`

the apostrophe will be mis-interpreted as a closing quote character. In

✗ `set xlabel "J\"org's data"`

the backslash before the " character, intended to be the latex control string for an umlaut (`\"o`), will instead be interpreted as a Pyxplot escape character. It will not be passed to latex, and a latex error will result. Whilst it is possible to write

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

this syntax is messy, as the backslashes are confusing to the eye. It is much neater to use (see Section 3.5 for an explanation of string escaping):

✔ `set xlabel r"""J\"org's data"""`

There are similar problems with

✗ `set xlabel e"2 \times 3"`

where the `\t` will be turned into a tab character, because extended escape characters are enabled. This string could be made legal by removing the `e` prefix (see Section 3.5).

3.6.1 Removing labels and titles

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
```

Query	Description
`all`	Lists all settings.
`axes`	Lists all of the currently configured axes.
`functions`	Lists all currently defined mathematical functions, both those which are built into Pyxplot and those which the user has defined.
`settings`	Lists the current values of all settings which can be set with the `set` command.
`units`	Lists all of the physical units which Pyxplot is currently set up to recognise.
`userfunctions`	Lists all current user-defined mathematical functions and subroutines.
`variables`	Lists the values of all currently-defined variables.

Table 3.3: The special keywords which the `show` command recognises.

The `unset` command may be followed by almost 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.

3.7 Querying the values of settings

As the previous section has demonstrated, the `set` command is used in a wide range of ways to configure the way in which plots appear; we will meet many more in due course. The corresponding `show` command can be used to query the current values of settings. To query the value of one particular setting, the `show` command should be followed by the name of the setting, as in the example:

```
show title
```

Alternatively, several settings may be queried at once, or all settings beginning with a certain string of characters can be listed, as in the following two examples:

```
show xlabel ylabel key
show g
```

The special keyword `settings` may be used to display the values of all settings which can be set with the `set` command. A list of other special keywords which the `show` command accepts is given in Table 3.3.

Generally, the `show` command displays each setting in the form of a typeable `set` command which could be used to set that setting, together with a comment to briefly explain what effect the setting has. This means that the output can be pasted directly into another Pyxplot terminal to copy settings from one session to another. However, some settings such as `papersize` are only pastable once the `set numerics typeable` command has been issued, for reasons which will be explained in Section 4.7.3.

When a color-highlighted interactive session is used, settings which have been changed are highlighted in yellow, whilst those settings which are unchanged

from Pyxplot's default configuration, or from a user-supplied configuration file, are shown in green.

3.8 Plotting data files

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

```
plot 'data.dat' using 3:5
```

This example would plot the contents of the fifth column of the file `data.dat` on the vertical axis, against the contents of the third column on the horizontal axis. As mentioned above, columns in data files can be separated by whitespace and/or commas. Algebraic expressions may also be used in place of column numbers, as in the example:

```
plot 'data.dat' using (3+$1+$2):(2+$3)
```

In such expressions, column numbers are 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 data file, plus three, on the horizontal axis, against two plus the value in the third column on the vertical axis. The column numbers in such expressions can also be replaced by algebraic expressions, and so `$2` can also be written as `$(2)` or `$(1+1)`. In the following example, the data points are all placed on the vertical line $x = 3$ – the brackets around the `3` distinguish it as a numerical constant rather than a column number – meanwhile their vertical positions are drawn from the value of some column n in the data file, where the value of n is itself read from the second column of the data file:

```
plot 'data.dat' using (3):$($2)
```

It is also possible to plot data from only selected lines within a data file. When Pyxplot reads a data file, it looks for any blank lines in the file. It divides the data file up into *data blocks*, each being separated from the next by a single blank line. The first data block is numbered 0, the next 1, and so on.

When two or more blank lines are found together, the data file is divided up into *index blocks*. The first index block is numbered 0, the next 1, and so on. Each index block may be made up of a series of data blocks. To clarify this, a labelled example data file is shown in Figure 3.1.

By default, when a data file 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 'data.dat' 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. To do so, the `every` modifier is used, which takes up to six values, separated by colons:

```
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 3.1: An example Pyxplot data file – the data file is shown in the two left-hand columns, and commands are shown to the right.

```
plot 'data.dat' every a:b:c:d:e:f
```

The values have the following meanings:

a	Plot data only from every a th line in data file.
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 are both valid statements:

```
plot 'data.dat' index 1 every 2:3
plot 'data.dat' index 1 every ::3
```

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

Comment lines may be included in data files by prefixing them with a hash character. Such lines are completely ignored by Pyxplot and do not count towards the one or two blank lines required to separate blocks and index blocks. It is often good practice to include comment lines at the top of data files to indicate their date and source. In Section 5.4 we will see that Pyxplot can read metadata from some comment lines which follow a particular syntax.

3.9 Plotting many data files at once

The wildcards `*` and `?` may be used in filenames supplied to the `plot` command to plot many data files at once. The following command, for example, plots

all data files in the current directory with a `.dat` suffix, using the same plot options:

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

In the graph's legend, full filenames are displayed, allowing the data files to be distinguished.

If a blank filename is supplied to the `plot` command, the last used data file is used again, as in the example:

```
plot 'data.dat' using 1:2, '' using 2:3
```

This can even be used with wildcards, as in the following example:

```
plot '*.dat' using 1:2, '' using 2:3
```

3.9.1 Horizontally arranged data files

Pyxplot also allows rows of data to be plotted against one another. To do so, the keyword `rows` is placed after the `using` modifier:

```
plot 'data.dat' index 1 using rows 1:2
```

For completeness, the syntax `using columns` is also accepted, specifying that columns should be plotted against one another, as happens by default:

```
plot 'data.dat' index 1 using columns 1:2
```

When plotting horizontally-arranged data files, the meanings of the `index` and `every` modifiers are altered slightly. The former continues to refer to vertically-displaced blocks of data separated by two blank lines. Blocks, as referenced in the `every` modifier, likewise continue to refer to vertically-displaced blocks of data points, 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 introduced in the previous section – now refer to vertical column numbers. For example,

```
plot 'data.dat' 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.

3.9.2 Choosing which data to plot

The final modifier which the `plot` command takes to allow the user to specify which subset(s) of a data file should be plotted is `select`. This can be used to plot only those data points in a data file which satisfy some given criterion, as in the following examples:

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

In the second example, two selection criteria are given, combined with the logical `and` operator. A full list of all of the operators recognised by Pyxplot, including logical operators, was given in Table 3.1.

When plotting using `lines` to connect the data points (see Section 3.13 for more information about Pyxplot's plotting styles), the default behaviour is for the lines not to be broken if a set of data points are removed by the `select` modifier. However, this behaviour is sometimes undesirable. To cause the plotted line to break when points are removed the `discontinuous` modifier is supplied after the `select` modifier, as in the example

```
plot sin(x) select ($2>0) discontinuous
```

which plots a set of disconnected peaks from the sine function.

3.10 The `replot` command

The `replot` command may be used to add more datasets to an existing plot, or to change its axis ranges. For example, having plotted one data file using the command

```
plot 'datafile1.dat'
```

another can be plotted on the same axes using the command

```
replot 'datafile2.dat' using 1:3
```

or the ranges of the axes on the original plot can be changed using the command

```
replot [0:1][0:1]
```

3.11 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 such as jpeg and png. The `set terminal` command[3] is used to specify the output format that is required, and the `set output` command is used to specify the file to which output should be directed. For example,

```
set terminal pdf
set output 'myplot.pdf'
plot 'datafile.dat'
```

would output a PDF plot of data from the file `datafile.dat` to the file `myplot.pdf`, which could be opened in Adobe Reader.

The `set terminal` command can also be used to configure various output options within each supported file format. For example, the following commands would produce black-and-white or color output respectively:

[3] Gnuplot users should note that the syntax of the `set terminal` command in Pyxplot is somewhat different from that used by Gnuplot; see Section 9.1.

```
set terminal monochrome
set terminal color
```

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

Both PostScript and 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 files in the Portable Document Format (PDF) read by Adobe Acrobat:

```
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 color
set terminal gif monochrome
```

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

```
set terminal x11
```

After changing terminals, the `refresh` command[4] is especially useful; it reproduces the last plot to have been generated in the newly-selected graphical format. For more details of the `set terminal` command, including how to produce gif and png images with transparent backgrounds, see Chapter 9.

3.12 Setting the size of output

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

```
set width 20
```

[4]The effect of the `refresh` command is very similar to that of the `replot` command with no arguments. The latter simply repeats the last `plot` command. We will see in Chapter 10 that the `refresh` command is to be preferred in the current context because it is applicable to vector graphics as well as to graphs.

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. $\left((1+\sqrt{5})/2\right)^{-1}$. The special command `set size square` sets the aspect ratio to unity (i.e. square).

3.13 Plotting styles

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

```
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, data files can be plotted with a line connecting the data points:

```
plot 'data.dat' with lines
```

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

```
plot 'data.dat' with yerrorbars
```

In this case, three columns of data need to be specified: the x- and y-coordinates of each data point, plus the size of the vertical error bar on that data point. By default, the first three columns of the data file are used, but as elsewhere (see Section 3.8), the `using` modifier can be used:

```
plot 'data.dat' using 2:3:7 with yerrorbars
```

Other plot styles supported by Pyxplot are listed in Section 8.2. More details of the `errorbars` plot style can be found in Section 8.2.2. Bar charts will be discussed in Section 8.2.4.

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, and are listed in Chapter 18.

The default plotting style, used when none is specified to the `plot` command, can also be changed. For example:

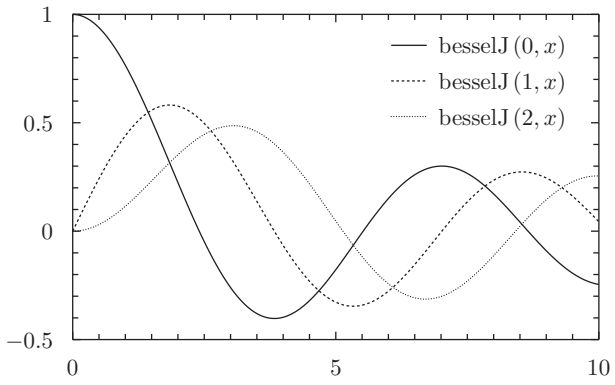
```
set style data lines
```

would change the default style used for plotting data from files to `lines`. Similarly, the `set style function` command changes the default style used when functions are plotted.

3.14 Setting axis ranges

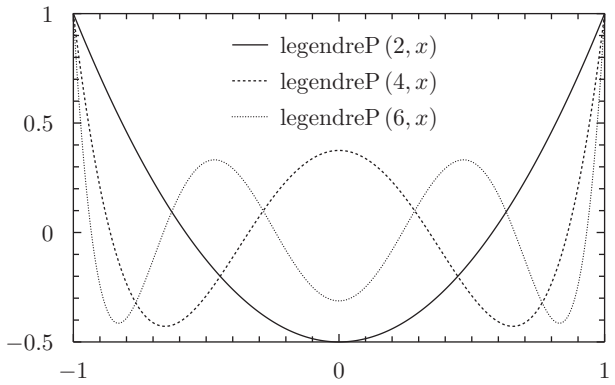
By default, Pyxplot automatically scales axes to some sensible range which contains all of the plotted data. However, it is possible for the user to override this and set his own range. This can be done directly from the `plot` command, by following the word `plot` with the syntax `[minimum:maximum]`.[5] The first specified range applies to the `x`-axis, and the second to the `y`-axis.[6] In the following example, the first three cylindrical Bessel functions are plotted in the range $0 < x < 10$:

```
plot [0:10][-0.5:1] besselJ(0,x), besselJ(1,x), besselJ(2,x)
```



Any of the values can be omitted, as in the following plot of three Legendre polynomials:

```
set key xcenter
plot [-1:1][:] legendreP(2,x), legendreP(4,x), legendreP(6,x)
```



[5] An alternative valid syntax is to replace the colon with the word `to`, i.e. `[minimum to maximum]`.

[6] As will be discussed in Section 8.8.1, if further ranges are specified, they apply to the `x2`-axis, then the `y2`-axis, and so forth.

Here, we have used the `set key` command to specify that the plot's legend should be horizontally aligned in the center of the plot, to complement the symmetry of the Legendre polynomials. This command will be described more fully in Section 8.7.

Alternatively, ranges can be set before the `plot` statement, using the `set xrange` command, as in the examples:

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

If an asterisk is supplied in place of either of the limits in this command, then any limit which had previously been set is switched off, and the axis returns to its default autoscaling behaviour:

```
set xrange [-2:*]
```

A similar effect may be obtained using the `set autoscale` command, which takes a list of the axes to which it is to apply. Both the upper and lower limits of these axes are set to scale automatically. If no list is supplied, then the command is applied to all axes.

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

The range supplied to the `set xrange` can be followed by the word `reverse` to indicate that the axis should run from right-to-left, or from top-to-bottom. In practice, this is of limited use when an explicit range is specified, as the following two commands are equivalent:

```
set xrange [-2:2] reverse
set xrange [2:-2] noreverse
```

However, this is useful when axes are set to autoscale:

```
set xrange [*:*] reverse
```

Axes can be set to have logarithmic scales by 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 8.8.1.

Example 1: A diagram of the trajectories of projectiles fired with different initial velocities.

In this example we produce a diagram of the trajectories of projectiles fired by a cannon at the origin with different initial velocities v and different angles of inclination θ. According to the equations of motion under constant acceleration, the distance of such a projectile from the origin after time t is given by

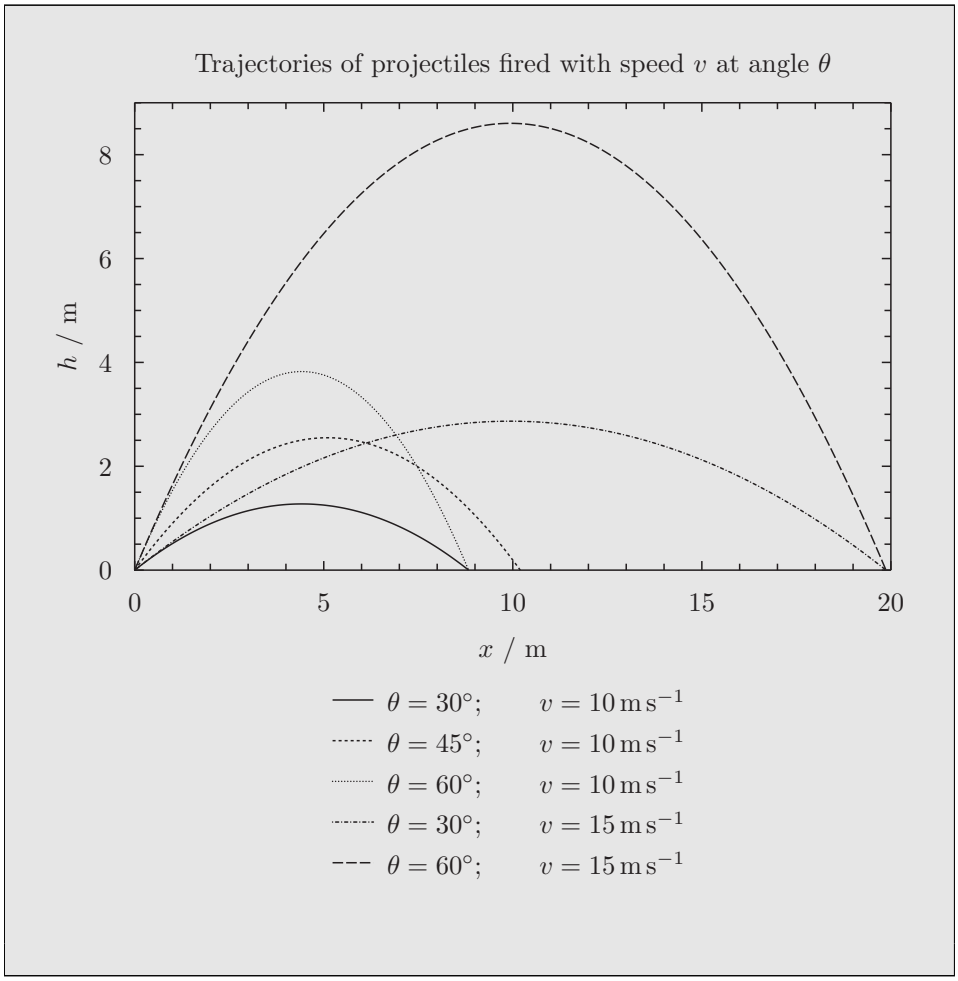
$$\begin{aligned} x(t) &= vt\cos\,\theta \\ h(t) &= vt\sin\,\theta + {}^1\!/_2 gt^2 \end{aligned}$$

where $x(t)$ is the horizontal displacement of the projectile and $h(t)$ the vertical displacement. Eliminating t from these equation gives the expression

$$h(x) = x\tan\,\theta - \frac{gx^2}{2v^2\cos^2\,\theta}.$$

In the script below, we plot this function for five different values of v and θ.

```
g   = phy.g     # Acceleration due to gravity
deg = unit(deg) # Convert degrees to radians

# The mathematical equation of a trajectory
h(x,theta,v) = x*tan(theta*deg) - 0.5*g*x**2/(v**2*cos(theta*deg)**2)

# Plot configuration
set xlabel r"$x$"
set ylabel r"$h$"
set xrange [unit(0*m):unit(20*m)]
set yrange [unit(0*m):]
set key below
set title r'Trajectories of projectiles fired with speed $v$ at angle $\theta$'
plot h(x,30,unit(10*m/s)) t r"$\theta=30^\circ;\qquad v=10\,{\rm m\,s^{-1}}$", \
h(x,45,unit(10*m/s)) t r"$\theta=45^\circ;\qquad v=10\,{\rm m\,s^{-1}}$", \
h(x,60,unit(10*m/s)) t r"$\theta=60^\circ;\qquad v=10\,{\rm m\,s^{-1}}$", \
h(x,30,unit(15*m/s)) t r"$\theta=30^\circ;\qquad v=15\,{\rm m\,s^{-1}}$", \
h(x,60,unit(15*m/s)) t r"$\theta=60^\circ;\qquad v=15\,{\rm m\,s^{-1}}$"
```



3.15 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 about which help is available. To display help on any given topic, type `help` followed by the name of the topic. For example,

```
help datafile
```

provides information on the format in which Pyxplot expects to read data files and

```
help plot
```

provides information about the `plot` command. Some topics have sub-topics, 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 set title
```

Chapter 4

Performing calculations

Often calculations need to be performed on data before they are plotted. This chapter and the next describe the mathematical environment which Pyxplot provides.

Most of the examples in this chapter act on single numerical values, displaying the results using the `print` command, demonstrating how Pyxplot may be used as a desktop calculator. The next chapter will extend this to use of Pyxplot's mathematical environment to analyse whole datasets and produce plots.

4.1 Variables

Variables can be assigned to hold numerical values using syntax of the form

```
a = 5.2 * sqrt(64)
```

which may optionally be written in longhand as

```
let a = 5.2 * sqrt(64)
```

Variables can subsequently be used by name in mathematical expressions, for example:

```
print a / sqrt(64)
```

Having been defined, variables can later be undefined – set to have no value – using syntax of the form:

```
a =
```

Variables can also hold non-numeric data, such as strings, colors, dates, lists and dictionaries. The syntax for defining many of these data structures is similar to that used by python, for example:

```
myList = [8,2,1,7]
myDict = {'john':27 , 'fred':14 , 'lisa':myList}
myDate = time.fromCalendar(2012,7,1,14,30,0)
```

More information about Pyxplot's data types can be found in Chapter 6.

A list of all of the variables which are currently defined can be obtained by typing `show variables`. Some constants are pre-defined by Pyxplot, and so a number of variables are listed even if none have been set by the user.

4.2 Physical constants

Many mathematical and physical constants are pre-defined in Pyxplot. A complete list of these can be found Chapter 14. Some of these, for example, `e`, `pi` and `goldenRatio` are standard mathematical constants which are accessible in the user's default namespace:

```
pyxplot> print pi
3.1415927
```

Others, such as physical constants, are of more specialist interest and are defined in modules. For example, the speed of light is defined in the physics module `phy`:

```
pyxplot> print phy.c
299792.46 km/s
```

Most of the pre-defined physical constants, such as this one, make use of Pyxplot's native ability to keep track of the physical units of quantities and to convert them between different unit systems – for example, between inches and centimeters. This will be explained in more detail in Section 4.6.

To list all of the functions and variables defined in a module such as `phy`, type

```
print phy
```

or simply

```
phy
```

4.3 Functions

Many standard mathematical and operating system functions are pre-defined within Pyxplot's mathematical environment. These range from everyday examples like trigonometric functions, to very specialised functions; there is even a function to return the phase of the Moon on any given day. As with the mathematical constant, common functions are defined in the user's default namespace, for example

```
pyxplot> print exp(2)
7.3890561
```

whilst others live in modules, for example

```
print ast.moonPhase( time.now() )
```

which returns the present phase of the Moon in radians, and

```
print os.path.filesize("/etc/passwd")
```

which returns the size of a file (in units of bytes, of course!).

A complete list of these functioned, sorted by module, can be found in Chapter 12. Another quick way to find out some more information about a function is the print the function object, for example:

```
pyxplot> print log
log(x) returns the natural logarithm of x.
```

All, of Pyxplot's built-in constants, functions and modules are contained in the module `defaults`, which can also be printed to view its contents:

```
print defaults
```

It is possible to access `pi`, for example, as `defaults.pi`, though in practice this syntax is very rarely needed. All of the objects in the `defaults` module are always accessible by name (i.e. they are always in any namespace), unless another local or global variable exists with the same name.

The user can define his own algebraic function definitions using a similar syntax to that used to declare new variables, as in the examples:

```
f()    = pi
g(x)   = x*sin(x)
h(x,y) = x*y
```

Function objects are just like any other variables, and can even be used as arguments to other functions:

```
pyxplot> f = sum
pyxplot> print f
sum(...) returns the sum of its arguments.
pyxplot> f = sin
pyxplot> g(x,y) = x(x(y))
pyxplot> print g(f,1)
0.74562414
```

User-defined functions can be undefined in the same way as any other variable, for example by typing:

```
f =
```

Where the logic required to define a particular function is greater than can be contained in a single algebraic expression, a subroutine should be used (see Section 7.8); these allow an arbitrary numbers of lines of Pyxplot code to be executed whenever a function is evaluated.

4.3.1 Spliced functions

The definitions of functions can be declared to be valid only within a certain domain of argument space, allowing for error checking when models are evaluated outside their domain of applicability. Furthermore, functions can be given

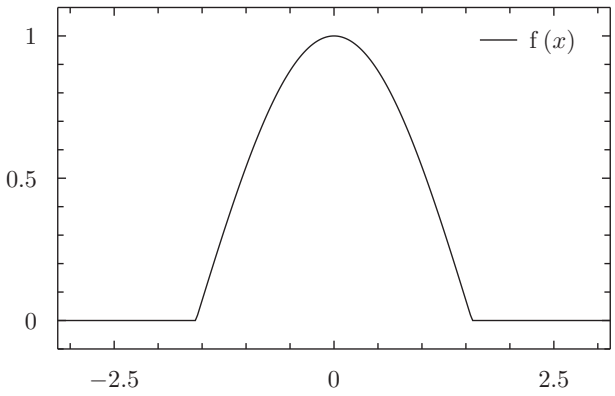
multiple definitions which are specified to be valid in different parts of argument space. We term this *function splicing*, since multiple algebraic definitions for a function are spliced together at the boundaries between their various domains. The following example would define a function which is only valid within the range $-\pi/2 < x < \pi/2$:[1]

```
truncated_cos(x)[-pi/2:pi/2] = cos(x)
```

Attempts to evaluate this function outside of the range in which it is defined would return an error that the function is not defined at the requested argument value. Thus, if the above function were to be plotted, no line would be drawn outside of the range $-\pi/2 < x < \pi/2$. A similar effect could also have been achieved using the `select` keyword (see Section 3.9.2). Sometimes, however, the desired behaviour is rather that the function should be zero outside of some region of parameter space where it has a finite value. This can be achieved as in the following example:

```
f(x) = 0
f(x)[-pi/2:pi/2] = cos(x)
```

Plotting this function would yield the following result:



To produce this function, we have made use of the fact that if there is an overlap in the domains of validity of multiple definitions of a function, then later declarations are guaranteed take precedence. The definition that the function equals zero is valid everywhere, but is overridden in the region $-\pi/2 < x < \pi/2$ by the second function definition.

Where functions have been spliced together, the `show functions` command will show all of the definitions of the spliced function, together with the regions of parameter space in which they are used. This is indicated using the same syntax that is used for defining spliced functions, such that the output can be stored and pasted into a future Pyxplot session to redefine exactly the same spliced function.

When a function takes more than one argument, multiple ranges can be specified, one for each argument. Any of the limits can be left blank if there is no upper or lower limit upon the value of that particular argument. In the following example, the function `f(a,b,c)` would only be defined when all of `a`, `b` and `c` were in the range $-1 \rightarrow 1$:

[1] The syntax `[-pi/2:pi/2]` can also be written `[-pi/2 to pi/2]`.

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

Function splicing can be used to define 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
```

Similar effects may also be achieved using the ternary conditional `?:` operator (see Section 7.7), for example:

```
f(x) = (x>0) ? 1 : 0
```

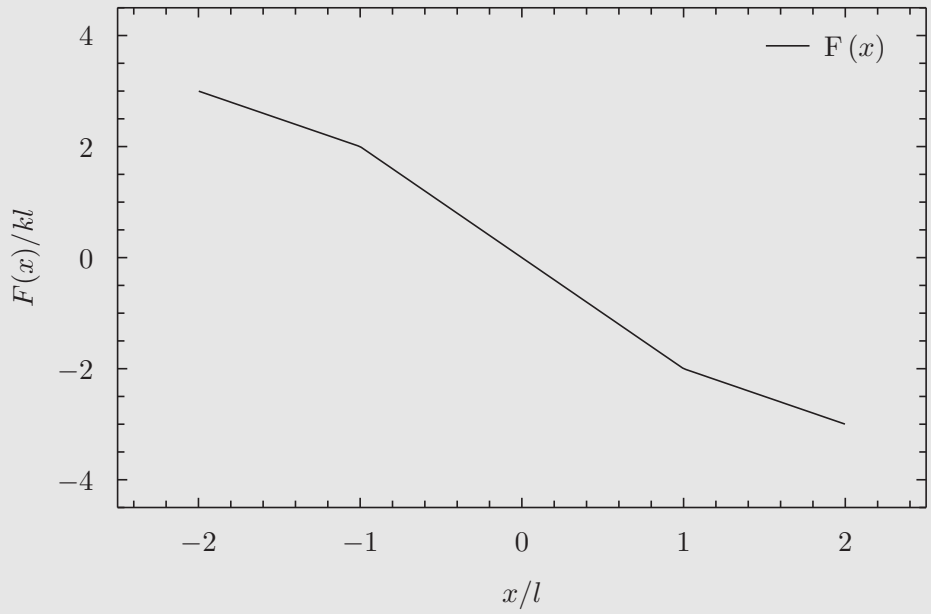
Example 2: Modelling a physics problem using a spliced function.

Question
A light bead is free to move from side to side between two walls which are placed at $x = -2l$ and $x = 2l$. It is connected to each wall by a light elastic string of natural length l, which applies a force $k\Delta x$ when extended by an amount Δx, but which applies no force when slack. What is the total horizontal force on the bead as a function of its horizontal position x?

Answer
This system has three distinct regimes. In the region $-l < x < l$, both strings are under tension. When $x < -l$, the left-hand string is slack, and only the right-hand string exerts a force. When $x > l$, the converse is true: only the left-hand string exerts a force. The case $|x| > 2l$ is not possible, as the bead would have to penetrate the hard walls. It is left as an exercise for the reader to use Hooke's Law to derive the following expression, but in summary, the force on the bead can be defined in Pyxplot as:

```
F(x)[-2*l :-  l] = -k*(x-l)
F(x)[-  l :   l] = -2*k*x
F(x)[   l : 2*l] = -k*(x+l)
```

where it is necessary to first define a value for `l` and `k`. Plotting these functions yields the result:



Attempting to plot this function with an `x`-axis which extends outside of the range of values of x for which $F(x)$ is defined, as above, will result in error messages being returned that the function could not be evaluated at all argument values. These can be suppressed by typing (see Section 4.4)

```
set numeric errors quiet
```

Example 3: Using a spliced function to calculate the Fibonacci numbers.

The Fibonacci numbers are defined to be the sequence of numbers in which each member is the sum of its two immediate predecessors, and the first three members of the sequence are $0, 1, 1$. Thus, the sequence runs $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$. In this example, we use function splicing to calculate the Fibonacci sequence in an iterative and highly inefficient way, hard-coding the first three members of the sequence and then using the knowledge that all of the subsequent members are the sums of their two immediate predecessors:

```
f(x)     = 0.0
f(x)[1:] = 1.0
f(x)[3:] = f(x-1) + f(x-2)
```

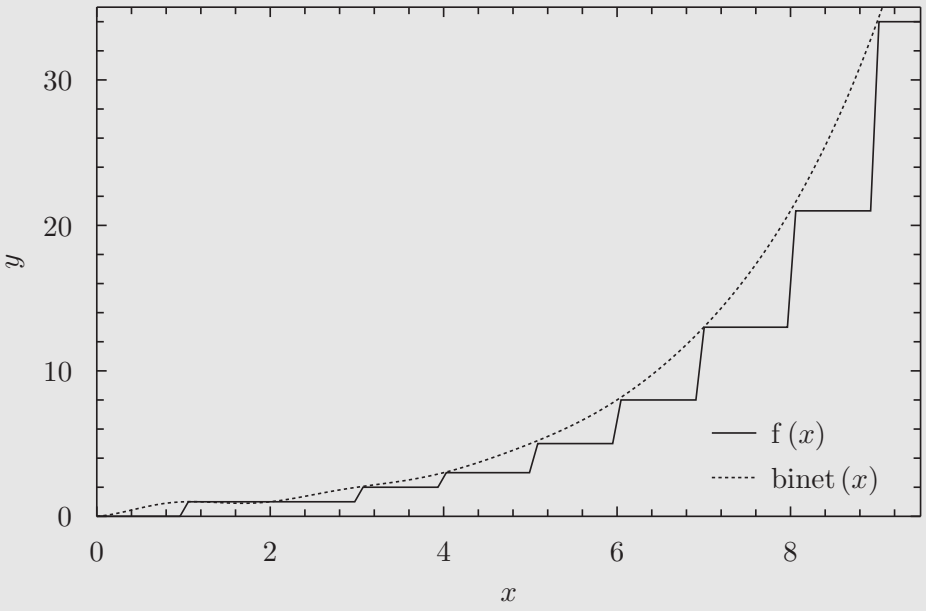
This method is highly inefficient because each evaluation spawns two further evaluations of the function, and so the number of operations required to evaluate `f(x)` scales as 2^x. It is inadvisable to evaluate it for $x \gtrsim 25$ unless you're prepared for a long wait.

A much more efficient method of calculating the Fibonacci numbers is to use Binet's formula,

$$f(x) = \psi^x - (1-\psi)^x\sqrt{5},$$

where $\psi = 1 + \sqrt{5}/2$ is the golden ratio, which provides an analytic expression for the sequence. In the following script, we compare the values returned by these two implementations. We enable complex arithmetic as Binet's formula returns complex numbers for non-integer values of x.

```
# Binet's Formula for the Fibonacci numbers
set numerics complex
binet(x) = Re((goldenRatio**x - (1-goldenRatio)**x) / sqrt(5))

set samples 100
set xrange [0:9.5]
set yrange [0:35]
set xlabel "$x$"
set ylabel "$y$"
set key bottom right
plot f(x) , binet(x)
```



4.4 Handling numerical errors

By default, an error message is returned whenever calculations return values which are infinite, as in the case of `1/0`, or when functions are evaluated outside the domain of parameter space in which they are defined, as in the case of `besseli(-1,1)`. Sometimes this behaviour is desirable: it flags up to the user

that a calculation has gone wrong, and exactly what the problem is. At other times, however, these error messages can be undesirable and may lead you to miss more genuine and serious errors buried in their midst.

For this reason, the issuing of explicit error messages when calculations return non-finite numeric results can be switched off by typing:

```
set numeric errors quiet
```

Having done this, expressions such as

```
x = besseli(-1,1)
```

fail silently, and variables which contain non-finite numeric results are displayed as `NaN`, which stands for *Not a Number*. The issuing of explicit errors may subsequently be re-enabled by typing:

```
set numeric errors explicit
```

Having turned off the display of numerical errors, it may be useful to use the `assert` command to throw an error message if a calculation has failed in an unrecoverable way that the user really ought to know about:

```
assert x>0 "Cannot continue with negative x"
```

The `assert` command should be followed by an algebraic expression which must be true for execution to continue. If it is false, an error results. Optionally, an error message can be included, as above, to tell the user what the problem is.

4.5 Working with complex numbers

In all of the examples given thus far, algebraic expressions have only been allowed to return real numbers: Pyxplot has not been handling any complex numbers. Since there are many circumstances in which the data being analysed may be known for certain to be real, complex arithmetic is disabled in Pyxplot by default. Expressions such as `sqrt(-1)` will return either an error or `NaN`. The most obvious example of this is the built-in variable `i`, which is set to equal `sqrt(-1)`:

```
pyxplot> print i
nan
```

Complex arithmetic may be enabled by typing

```
set numeric complex
```

and then disabled again by typing

```
set numeric real
```

Once complex arithmetic has been enabled, many of Pyxplot's built-in mathematical functions accept complex input arguments, including the logarithm function, all of the trigonometric functions, and the exponential function. A complete list of functions which accept complex inputs can be found in Appendix 12.

Complex number literals can be entered into algebraic expressions in either of the following two forms:

```
print (2 + 3*i       )
print (2 + 3*sqrt(-1))
```

The former version depends upon the pre-defined system variable `i` being defined to equal $\sqrt{-1}$. The user could cause this to stop working, of course, by redefining this variable to have a different value. However, in this case the variable `i` could straightforwardly be returned to its default value by typing:

```
i=sqrt(-1)
```

The user can, of course, define any other variable to equal $\sqrt{-1}$, thus allowing him to use any other letter, e.g. `j`, to represent the imaginary component of a number.

Several built-in functions are provided for manipulating complex numbers. The `Re(z)` and `Im(z)` functions return respectively the real and imaginary parts of a complex number z. The `arg(z)` function returns the complex argument of z. And the `abs(z)` function returns the modulus of z. The `conjugate(z)` command returns the complex conjugate of z. The following lines of code demonstrate the use of these functions:

```
pyxplot> set numeric complex
pyxplot> x=0.5
pyxplot> print Re(exp(i*x))
0.87758256
pyxplot> print cos(x)          # This equals the above
0.87758256
pyxplot> print arg(exp(i*x)) # This equals x
0.5
```

4.6 Working with physical units

Pyxplot, and all its mathematical functions, have native support for numbers to have physical units. This means, for example, that multiplying two lengths yields an area, and taking passing a selection of lengths to the `max(...)` function returns the longest of the lengths supplied.

This makes it a powerful desktop tool for converting measurements between different systems of units – for example, between imperial and metric units – or for doing physical calculations.

The special function `unit()` is used to specify the physical unit associated with a quantity. For example, the expression

```
print 2*unit(s)
```

takes the number 2 and multiplies it by the unit `s`, which is the SI abbreviation for seconds. Technically, the function `unit(s)` returns a numeric object equal to one second.

The resulting quantity above, which has dimensions of time, could then, for example, be divided by the unit `hr` to find the dimensionless number of hours in two seconds:

```
print 2*unit(s)/unit(hr)
```

Compound units such as miles per hour, which is defined in terms of two other units, can be used as in

```
print 2*unit(miles/hour)
```

In many cases, commonly-used units such units have their own explicit abbreviations, in this case `mph`:

```
print 2*unit(mph)
```

As these examples demonstrate, the `unit()` function can be passed a string of units either multiplied together with the `*` operator, or divided using the `/` operator. Units may be raised to powers with the `**` operator[2], as in the example:

```
pyxplot> a = 2*unit(m**2)
pyxplot> print "An area of %f square feet"%(a/unit(ft**2))
An area of 21.527821 square feet
```

As the examples above have demonstrated, units may be referred to by either their abbreviated or full names, and each of these may be used in either their singular or plural forms. For example, `s`, `second`, and `seconds` are all valid and equivalent. A complete list of all of the units which Pyxplot recognises by default, together with all of their recognised names, can be found in Appendix 15.

SI units may be preceded with SI prefixes, as in the examples[3]:

```
a = 2*unit(um)
a = 2*unit(micrometers)
```

When quantities with physical units are substituted into algebraic expressions, Pyxplot automatically checks that the expression is dimensionally correct before evaluating it. For example, the following expression is not dimensionally correct and would return an error because the first term in the sum has dimensions of velocity, whereas the second term is a length:

✗
```
a = 2*unit(m)
b = 4*unit(s)
print a/b + a
```

[2]The `^` character may be used as an alias for the `**` operator, though this notation is arguably confusing, since the same character is used for the binary exclusive or operator in Pyxplot's normal arithmetic.

[3]As the first of these examples demonstrates, the letter `u` is used as a Roman-alphabet substitute for the Greek letter μ.

Pyxplot continues to throw an error in this case, even when explicit numerical errors are turned off with the `set numeric errors quiet` command, since it is deemed a serious error: the above expression would never be correct for any values of `a` and `b` given their dimensions.

A large number of units are pre-defined in Pyxplot by default, a complete list of which can be found in Appendix 15. However, the need may occasionally arise to define new units. It is not possible to do this from an interactive Pyxplot terminal, but it is possible to do so from a configuration script which Pyxplot runs upon start-up. Such configuration scripts will be discussed in Chapter 19. New units may either be derived from existing SI units, alternative measures of existing quantities, or entirely new base units such as numbers of CPU cycles or man-hours of labour.

4.6.1 Treatment of angles in Pyxplot

There are a small number of cases where Pyxplot's handling of units does not completely follow normal (i.e. SI) conventions, which require further explanation. The most obvious such case is its handling of angles.

By convention, the SI system of units does not have a base unit of angle: instead, the radian is considered to be a dimensionless unit. There are some strong mathematical reasons why this makes sense, since it makes it possible to write equations such as

$$d = \theta r$$

and

$$x = \exp(a + i\theta),$$

which would otherwise have to be written as, for example,

$$d = 2\pi \left(\frac{\theta}{2\pi\,\mathrm{rad}}\right) r = \left(\frac{\theta}{\mathrm{rad}}\right) r$$

in order to be strictly dimensionally correct.

However, it also has some disadvantages since some physical quantities such as fluxes per steradian are measured per unit angle or per unit solid angle, and the SI system traditionally[4] offers no way to dimensionally distinguish these from one another or from quantities with no angular dependence. In addition, many of Pyxplot's vector graphics commands take rotation angles as inputs, and it is useful to express these in units of angle.

In most cases, the user is free to decide whether angles should have units. All of the following print statements are equivalent:

```
print sin(pi)
print sin(180*unit(deg))
print sin(pi *unit(rad))
print sin(0.5*unit(rev))
```

However, it is useful to be able to define whether inverse trigonometric functions such as `asin(x)` and `atan(x)` return results with units of angle, or which are dimensionless. By default, these functions return dimensionless results, but this may be changed using the commands:

[4] Radians are sometimes treated in the SI system as *supplementary* or derived units.

```
set unit angle dimensionless
set unit angle nodimensionless
```

Note that even when inverse trigonometric functions are set to return dimensionless outputs, expressions such as `unit(rad)+1` are still dimensionally incorrect. Functions such as `sin(x)` and `exp(x)` can always accept inputs which are either dimensionless, or have units of angle.

4.6.2 Converting between different temperature scales

Pyxplot can convert temperatures between different temperature scales, for example between °C, °F and K. However, these conversions have some subtleties which are unique to temperature conversions. This means they should be used with some caution.

Consider the following two questions:

- How many Kelvin corresponds to a temperature of 20°C?
- How many Kelvin corresponds to a temperature *rise* of 20°C?

The answers to these two questions are 293 K and 20 K respectively: although we are converting from 20°C in both cases, the corresponding number of Kelvin depends whether we are talking about an *absolute* temperature or a *relative* temperature. A heat capacity of 1 J/°C equals 1 J/K, even though a temperature of 1°C does not equal a temperature of 1 K.

The cause of this problem, and the reason why it rarely affects any physical units other than temperatures is that there exists such a thing as absolute temperature.

Take the example of distances. Distances are almost always relative: they measure distance gaps between points. Occasionally people might choose to express positions as distance from some particular origin. But if scheme A involved measuring in meters from New York, and scheme B involved measuring in feet from Chicago, they wouldn't expect Pyxplot to convert between the two systems.

The problem of converting between temperature systems is just like this. One system measures distance in degrees Fahrenheit away from 0°F; another the distance in degrees Celsius away from 0°C.[5]

As Pyxplot cannot distinguish between absolute and relative temperatures, it takes a safe approach of performing algebra consistently with any unit of temperature, never performing automatic conversions between different temperature scales. A calculation based on temperatures measured in °F will produce an answer measured in °F. However, as converting temperatures between temperature scales is a useful task which is often wanted, this is allowed, when specifically requested, in the specific case of dividing one temperature by another unit of temperature to get a dimensionless number, as in the following example:

[5]There is one other common example of this problem. Times are expressed as absolute quantities when dates are written down. The year AD 1453 implicitly corresponds to a period of 1453 years after the Christian epoch. So, similar problems arise when trying to convert such a year into the Muslim calendar, which counts from the year AD 622. Pyxplot can, incidentally, make this conversion, using date objects, as will be seen in Section 4.11.

✔
```
print 98*unit(oF) / unit(oC)
```

Note that the two units of temperature must be placed in separate `unit(...)` functions. The following is not allowed:

✗
```
print 98*unit(oF / oC)
```

Note that such a conversion always assumes that the temperatures supplied are **absolute** temperatures. Pyxplot has no facility for converting relative temperatures between different scales. This must be done manually.

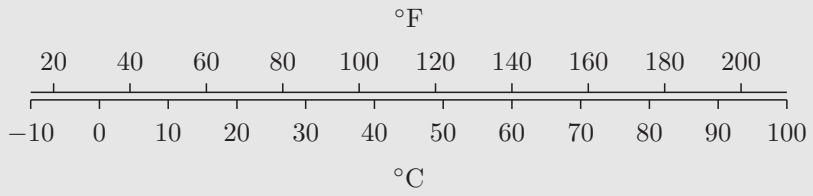
The conversion of derived units of temperature, such as J/K or $^\circ C^2$, to derived units of other temperature scales, such as J/°F or K^2, is not permitted, since in general these conversions are ill-defined.

The moral of this story is: pick what unit of temperature you want to work in, convert all of your temperatures to that scale, and then stick to it.

Example 4: Creating a simple temperature conversion scale.

In this example, we use Pyxplot's automatic conversion of physical units to create a temperature conversion scale.

```
set size ratio 1e-2
set axis x2 linked x using x*unit(oC)/unit(oF)
set axis y invisible
set xtics outward -10,10
set x2tics outward 20,20
set xlabel r"$^\circ$C"
set x2label r"$^\circ$F"
plot [-10:100]
```



4.7 Configuring how numbers are displayed

4.7.1 Display of physical units

When displaying quantities that have physical units, Pyxplot searches through its database of known units looking for the most appropriate unit, or combination of units, to use. By default, SI units, or combinations of SI units, are chosen for preference, and SI prefixes such as milli- or kilo- are applied where appropriate. This behaviour can, however, be extensively configured.

Name	Description
`ancient`	Ancient units, especially those used in the Authorised Version of the Bible.
`CGS`	CGS units.
`Imperial`	British imperial units.
`Planck`	Planck units, also known as natural units, which make several physical constants equal unity.
`SI`	SI units.
`US`	US customary units.

Table 4.4: A list of Pyxplot's unit schemes.

The most general configuration option allows one of several *units schemes* to be selected, each of which comprises a list of units which are deemed to be members of the particular scheme. For example, in the CGS unit scheme, all lengths are displayed in centimeters, all masses are displayed in grammes, all energies are displayed in ergs, and so forth. In the imperial unit scheme, quantities are displayed in British imperial units – inches, pounds, pints, and so forth. In the US unit scheme, US customary units are used. The current unit scheme can be changed using the `set unit scheme` command:

```
pyxplot> vol = 3*unit(m**3)
pyxplot> set unit scheme si ; print vol
3 m**3
pyxplot> set unit scheme cgs ; print vol
3000000 cm**3
pyxplot> set unit scheme imperial ; print vol
3.9238519 yd**3
pyxplot> set unit scheme us ; print vol
12680.259 cups_US
```

A complete list of Pyxplot's unit schemes can be found in Table 4.4.

These units schemes are often sufficient to ensure that most quantities are displayed in the desired units, but commonly there are a few specific quantities in any particular piece of work where non-standard units are used. For example, a study of Jupiter-like planets might express masses in Jupiter masses, rather than kilograms. A study of the luminosities of stars might express powers in units of solar luminosities, rather than watts. And a cosmology paper might express distances in megaparsecs. This level of control is made available through the `set unit of` command. The three examples just given could be achieved using the following commands:

```
set unit of mass Mjupiter
set unit of power solar_luminosity
set unit of length parsec
```

An astronomer wishing to express masses in Pluto masses would need to first define the Pluto mass as a user-defined unit, since it is not pre-defined unit within Pyxplot. In Chapter 19, we shall see how to define new units in a configuration script. Having done so, the following syntax would be allowed:

```
set unit of mass Mpluto
```

The `set unit preferred` command offers a slightly more flexible way of achieving the same result. Whereas the `set unit of` command can only operate on named quantities such as lengths, areas and powers, and cannot act upon compound units such as `W/Hz`, the `set unit preferred` command can act upon any unit or combination of units:

```
set unit preferred parsec
set unit preferred W/Hz
set unit preferred N*m
```

The latter two examples are particularly useful when working with spectral densities (powers per unit frequency) or torques (forces multiplied by distances). Unfortunately, both of these units are dimensionally equal to energies, and so are displayed by Pyxplot in joules by default. The above statement overrides such behaviour. Having set a particular unit to be preferred, this can be unset as in the following example:

```
set unit nopreferred parsec
```

By default, units are displayed in their abbreviated forms, for example `A` instead of `amperes` and `W` instead of `watts`. Furthermore, SI prefixes such as milli- and kilo- are applied to SI units where they are appropriate. Both of these behaviours can be turned on or off, in the former case with the commands

```
set unit display abbreviated
set unit display full
```

and in the latter case using the following pair of commands:

```
set unit display prefix
set unit display noprefix
```

4.7.2 Changing the accuracy to which numbers are displayed

By default, when numbers are displayed, they are printed accurate to eight significant figures, although fewer figures may actually be displayed if the final digits are zeros or nines.

This is generally a helpful convention: Pyxplot's internal arithmetic is generally accurate to around 16 significant figures, and so it is quite conceivable that a calculation which is supposed to return, say, 1, may in fact return 0.999 999 999 999 999 9. Likewise, when complex arithmetic is enabled, routines which are expected to return real numbers may in fact return results with imaginary parts at the level of one part in 10^{16}. By displaying numbers to only eight significant figures in such cases, the user is usually shown the 'right' answer, instead of a noisy and unattractive one.

However, there may also be cases where more accuracy is desirable, in which case, the number of significant figures to which output is displayed can be set using the command

```
n = 12
set numerics sigfig n
```

where `n` can be any number in the range 1-30. It should be noted that the number supplied is the *minimum* number of significant figures to which numbers are displayed; on occasion an extra figure may be displayed.

Alternatively, the string substitution operator, described in Section 6.2.1 may be used to specify how a number should be displayed on a one-by-one basis, as in the examples:

```
pyxplot> print "%d"  %(pi) # Print the integer part of pi
3
pyxplot> print "%.5f"%(pi) # Print pi in non-scientific format, to 5 d.p.
3.14159
pyxplot> print "%.5e"%(pi) # Print pi in scientific format, to 5 d.p.
3.14159e+00
pyxplot> print "%s"  %(pi) # Print pi as normal
3.1415927
```

4.7.3 Creating pastable text

Pyxplot's default convention of displaying numbers in a format such as

```
(2+3i) meters
```

is well-suited for creating text which is readable by human users, but is less well-suited for creating text which can be copied and pasted into another calculation in another Pyxplot terminal, or for creating text which could be used in a latex text label on a plot. For this reason, the `set numerics display` command allows the user to choose between three different ways in which numbers can be displayed:

```
pyxplot> set numerics display natural
pyxplot> print phy.c
299792.46 km/s
pyxplot> set numerics display typeable
pyxplot> print phy.c
299792.46*unit(km/s)
pyxplot> set numerics display latex
pyxplot> print phy.c
$299792.46\,\mathrm{km}/\mathrm{s}$
```

The first case is the default way in which Pyxplot displays numbers. The second case produces text which forms a valid algebraic expression which could be pasted into another Pyxplot calculation. The final case produces a string of latex text which could be used as a label on a plot.

4.8 Numerical integration and differentiation

Two special functions, `int_dx()` and `diff_dx()`, may be used to integrate or differentiate algebraic expressions numerically. In each case, the letter `x` is the

dummy variable which is to be used in the integration or differentiation and may be replaced by any valid variable name of up to 16 characters in length.

The function `int_dx()` takes three parameters – firstly the expression to be integrated, which may optionally be placed in quotes, followed by the minimum and maximum integration limits. These may have any physical dimensions, so long as they match, but must both be real numbers. 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 one further optional parameter. The first is the expression to be differentiated, which, as above, may optionally placed in quotes for clarity. This should be followed by the numerical value x of the dummy variable at the point where the expression is to be differentiated. This value may have any physical dimensions, and may be a complex number if complex arithmetic is enabled. The final, optional, parameter to the `diff_dx()` function is an approximate step size, which indicates the range of argument values over which Pyxplot should take samples to determine the gradient. If no value is supplied, a value of $10^{-6}x$ is used, replaced by 10^{-6} if $x = 0$. 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)
```

When complex arithmetic is enabled, Pyxplot checks that the function being differentiated satisfies the Cauchy-Riemann equations, and returns an error if it does not, to indicate that it is not differentiable. The following is an example of a function which is not differentiable, and which throws an error because the Cauchy-Riemann equations are not satisfied:

```
set num comp
print diff_dx(Re(sin(x)),1)
```

Advanced users may be interested to know that `int_dx()` function is implemented using the `gsl_integration_qags()` function of the Gnu Scientific Library (GSL), and the `diff_dx()` function is implemented using the `gsl_deriv_central()` function of the same library. Any caveats which apply to the use of these routines also apply to Pyxplot's numerical calculus.

Example 5: Integrating the function $\mathrm{sinc}(x)$.

The function $\mathrm{sinc}(x)$ cannot be integrated analytically, but it can be shown that

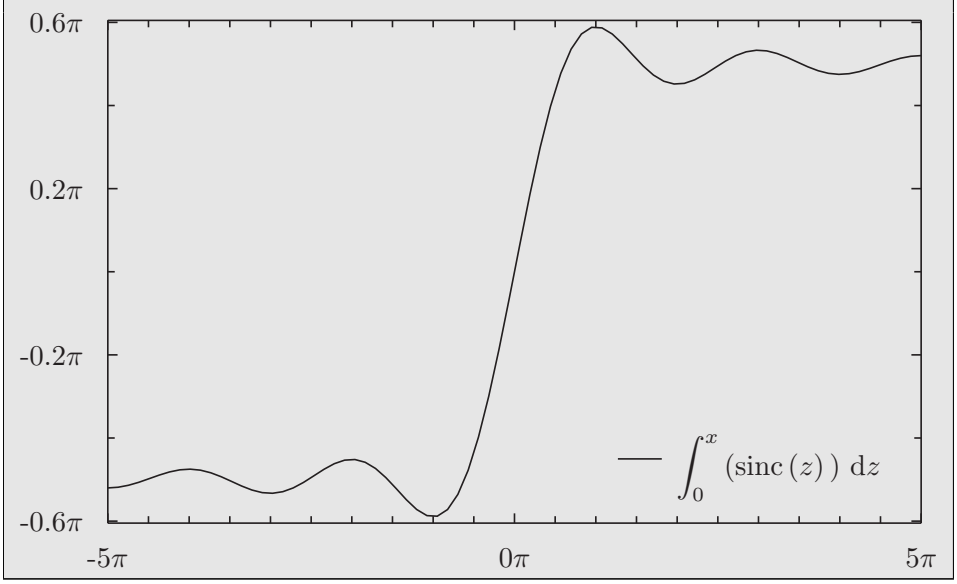
$$\int_0^{\pm\infty} \mathrm{sinc}(x)\,\mathrm{d}x = \pm\pi/2.$$

In the following script, we use Pyxplot's facilities for numerical integration to produce a plot of

$$y = \int_0^x \mathrm{sinc}(x)\,\mathrm{d}x.$$

We reduce the number of samples taken along the abscissa axis to 80, as evaluation of the numerical integral may be time consuming on older computers. We use the `set xformat` command (see Section 8.8.8) to demark both the `x`- and `y`-axes in fractions of π:

```
set samples 80
set key bottom right
set xformat r"%s$\pi$"%(x/pi)
set yformat r"%s$\pi$"%(y/pi)
set xrange [-5*pi:5*pi]
plot int_dz(sinc(z),0,x)
```



4.9 Solving systems of equations

The `solve` command can be used to solve systems of one or more simultaneous equations numerically. It takes as its arguments a comma-separated list of the equations which are to be solved, and a comma-separated list of the variables which are to be found. The latter should be prefixed by the word `via`, to separate it from the list of equations:

```
solve <equation 1>, ... via <variable 1>, ...
```

Note that the time taken by the solver dramatically increases with the number of variables which are simultaneously found, whereas the accuracy achieved simultaneously decreases. The following example solves a simple pair of simultaneous equations of two variables:

```
pyxplot> solve x+y=10, x-y=3 via x,y
pyxplot> print x
6.5
pyxplot> print y
3.5
```

No output is returned to the terminal if the numerical solver succeeds, otherwise an error message is displayed. If any of the fitting variables are already defined prior to the `solve` command's being called, their values are used as initial guesses, otherwise an initial guess of unity for each fitting variable is assumed. Thus, the same `solve` command returns two different values in the following two cases:

```
pyxplot> x= # Undefine x
pyxplot> solve cos(x)=0 via x
pyxplot> print x/pi
0.5
pyxplot> x=10
pyxplot> solve cos(x)=0 via x
pyxplot> print x/pi
3.5
```

In cases where any of the variables being solved for are not dimensionless, it is essential that an initial guess with appropriate units be supplied, otherwise the solver will try and fail to solve the system of equations using dimensionless values:

✗
```
x =
y = 5*unit(km)
solve x=y via x
```

✔
```
x = unit(m)
y = 5*unit(km)
solve x=y via x
```

The `solve` command works by minimising the squares of the residuals of all of the equations supplied, and so even when no exact solution can be found, the best compromise is returned. The following example has no solution – a system of three equations with two variables is over-constrained – but Pyxplot nonetheless finds a compromise solution:

```
pyxplot> solve x+y=10, x-y=3, 2*x+y=16 via x,y
pyxplot> print x
6.4220634
pyxplot> print y
3.4266948
```

When complex arithmetic is enabled, the `solve` command allows each of the variables being fitted to take any value in the complex plane, and thus the number of dimensions of the fitting problem is effectively doubled – the real and imaginary components of each variable are solved for separately – as in the following example:

```
pyxplot> set numerics complex
pyxplot> solve exp(x)=e*i via x
pyxplot> print Re(x)
-1227.7
pyxplot> print Im(x)/pi
0
```

4.10 Searching for minima and maxima of functions

The `minimize` and `maximize` commands can be used to find the minima or maxima of algebraic expressions. In each case, a single algebraic expression should be supplied for optimisation, together with a comma-separated list of the variables with respect to which it should be optimised. In the following example, a minimum of the sinusoidal function $\cos(x)$ is sought:

```
pyxplot> set numerics real
pyxplot> x=0.1
pyxplot> minimize cos(x) via x
pyxplot> print x/pi
1
```

Note that this particular example doesn't work when complex arithmetic is enabled, since $\cos(x)$ diverges to $-\infty$ at $x = \pi + \infty i$.

Various caveats apply both to the `minimize` and `maximize` commands, as well as to the `solve` command. All of these commands operate by searching numerically for optimal sets of input parameters to meet the criteria set by the user. As with all numerical algorithms, there is no guarantee that the *locally* optimum solutions returned are the *globally* optimum solutions. It is always advisable to double-check that the answers returned agree with common sense.

These commands can often find solutions to equations when these solutions are either very large or very small, but they usually work best when the solution they are looking for is roughly of order unity. Pyxplot does have mechanisms which attempt to correct cases where the supplied initial guess turns out to be many orders of magnitude different from the true solution, but it cannot be guaranteed not to wildly overshoot and produce unexpected results in such cases. To reiterate, it is always advisable to double-check that the answers returned agree with common sense.

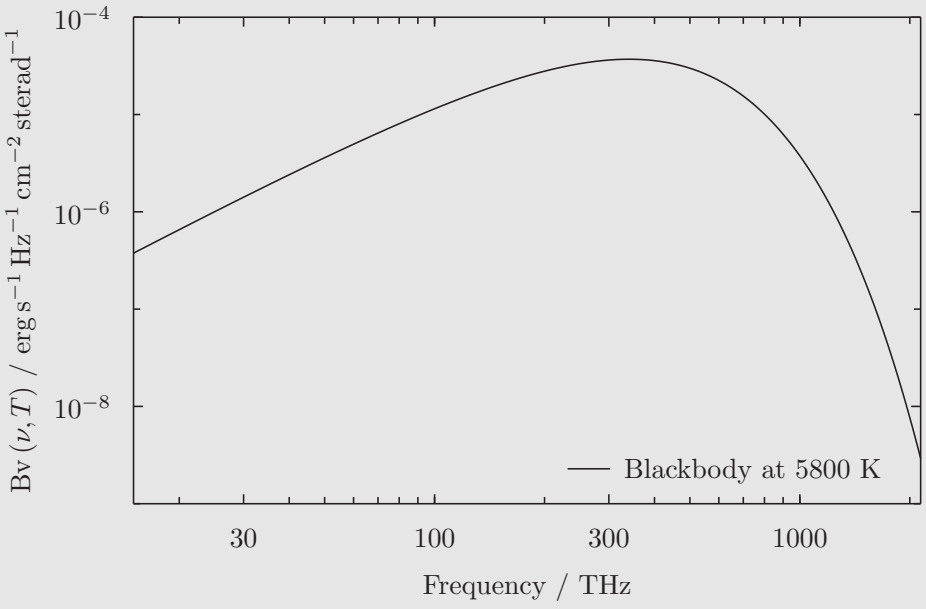
Example 6: Finding the maximum of a blackbody curve.

When a surface is heated to any given temperature T, it radiates thermally. The amount of electromagnetic radiation emitted at any particular frequency, per unit area of surface, per unit frequency of light, is given by the Planck Law:

$$B_\nu(\nu, T) = \left(\frac{2h^3}{c^2}\right) \frac{\nu^3}{\exp(h\nu/kT) - 1}$$

The visible surface of the Sun has a temperature of approximately 5800 K and radiates in such a fashion. In this example, we use the `solve`, `minimize` and `maximize` commands to locate the frequency of light at which it emits the most energy per unit frequency interval. This task is simplified as Pyxplot has a system-defined mathematical function `Bv(nu,T)` which evaluates the expression given above.

Below, a plot is shown of the Planck Law for $T = 5800\,\mathrm{K}$ to aid in visualising the solution to this problem:



To search for the maximum of this function using the `maximize` command, we must provide an initial guess to indicate that the answer sought should have units of Hz:

```
pyxplot> nu = 1e14*unit(Hz)
pyxplot> maximize phy.Bv(nu,5800*unit(K)) via nu
pyxplot> print nu
340.9781 THz
```

This maximum could also be sought be searching for turning points in the function $B_\nu(\nu, T)$, i.e. by solving the equation

$$\frac{\mathrm{d}B_\nu(\nu,T)}{\mathrm{d}\nu} = 0.$$

This can be done as follows:

```
pyxplot> nu = 2e14*unit(Hz)
pyxplot> solve diff_dv(phy.Bv(v,5800*unit(K)),nu) = \
.......>         unit(0*W/Hz**2/m**2/sterad) via nu
pyxplot> print nu
340.9781 THz
```

Finally, this maximum could also be found using Pyxplot's built-in function `Bvmax(T)`:

```
pyxplot> print phy.Bvmax(5800*unit(K))
340.97806 THz
```

4.11 Working with time-series data

Time-series data need to be handled carefully. If times and dates are specified in local time, then conversions may be necessarily between timezones, especially around the beginning and end of daylight saving time.

Even when this it not an issue, months have different lengths and leap years have an extra day, which mean it is not straightforward to convert a series of calendar dates into elapsed times between the data points.

On a more basic level, even time expressed in hours and minutes are complicated by being expressed as non-decimal fractions of days.

To simplify the process of working with dates and times, Pyxplot has native `date` object type, together with pre-defined functions in the `time` module for creating and manipulating such objects. A date object represents a specific moment in time, and can be created from a time and date specified in any arbitrary timezone. It is then possible to read out the time and date components of this date object in any other arbitrary timezone.

The functions for creating `date` objects are as follows:

```
time.fromCalendar(year,month,day,hour,min,sec,<timezone>)
```

This function creates a date object from the specified calendar date. It takes six compulsary numerical inputs: the year, the month number (1–12), the day of the month (1–31), the hour of day (0–24), the number of minutes (0–59), and the number of seconds (0–59). To enter dates before AD 1, a year of 0 should be passed to indicate 1 BC, −1 should be passed to indicate the year 2 BC, and so forth.

A timezone may optionally be specified as the final argument to the function. If no timezone is specified, then the default is used, which may be set using the `set timezone` command. The timezone should be specified as a location string, of the form `Europe/London`, `America/New_York` or `Australia/Perth`, as used by the `tz database`. A complete list of available timezones can be found here:

http://en.wikipedia.org/wiki/List_of_tz_database_time_zones.

Daylight saving time will be applied as appropriate for the specified location. Note that strings such as `GMT`, `EDT` or `CEST` are *not* allowed as timezones; a *location* should be specified.

If universal time is used, the timezone may be specified as `UTC`.

`time.fromUnix(t)`

This function creates a date object from the specified numerical Unix time – i.e. the number of seconds ellapsed since midnight on 1st January 1970 UTC.

`time.fromJD(t)`

This function creates a date object from the specified numerical Julian date.

`time.fromMJD(t)`

This function creates a date object from the specified numerical modified Julian date.

`time.now()`

This function takes no arguments, returns a `date` object corresponding to the current system clock time, as in the following example:

```
pyxplot> print time.now()
Tue 2012 Sep 4 20:57:00 UTC
pyxplot> set timezone "America/Los_Angeles"
pyxplot> print time.now()
Tue 2012 Sep 4 13:57:41 PDT
```

Note that the date object created by the `time.now()` function is identical regardless of timezone, but it is *displayed* differently depending upon the current timezone.

The following example creates a date object representing midnight on 1st January 2000, in universal time, and in Western Australian time:

```
pyxplot> print time.fromCalendar(2000,1,1,0,0,0)
Sat 2000 Jan 1 00:00:00 UTC
pyxplot> a = time.fromCalendar(2000,1,1,0,0,0,"Australia/Perth")
pyxplot> print a # Note that this does not use Australian time
Fri 1999 Dec 31 15:59:59 UTC
pyxplot> set timezone "Pacific/Chatham"
pyxplot> print a
Sat 2000 Jan 1 05:45:00 +1345
pyxplot> set timezone "Antarctica/South_Pole"
pyxplot> print a
Sat 2000 Jan 1 05:00:00 NZDT
pyxplot> print a.toYear() # at the south pole
2000
pyxplot> print a.toYear("Europe/London")
1999
```

Once created, it is possible to add numbers with physical units of time to

dates, as in the following example:

```
pyxplot> myDate = time.fromCalendar(2012,8,1,0,0,0)
pyxplot> print myDate + unit(7*day)
Wed 2012 Aug 8 00:00:00 UTC
pyxplot> print myDate - unit(2000*day)
Fri 2007 Feb 9 00:00:00 UTC
```

Standard string representations of calendar dates can be produced with the `print` command. It is also possible to use the string substitution operator, as in `"%s"%(date)`, or the `str` method of `date` objects, as in `date.str()`.

In addition, the `time.string` function can be used to choose a custom display format for the date, or to specify the timezone in which the date should be displayed. Its arguments are as follows:

```
time.string(t,<format>,<timezone>)
```

This function returns a string representation of the specified date object t. The second argument is optional, and may be used to control the format of the output. If no format string is provided, then the format
`"%a %Y %b %d %H:%M:%S %Z"`
is used. In such format strings, the following tokens are substituted for various parts of the date:

Token	Value
`%%`	A literal % sign.
`%a`	Three-letter abbreviated weekday name.
`%A`	Full weekday name.
`%b`	Three-letter abbreviated month name.
`%B`	Full month name.
`%C`	Century number, e.g. 21 for the years 2000-2099.
`%d`	Day of month.
`%H`	Hour of day, in range 00-23.
`%I`	Hour of day, in range 01-12.
`%k`	Hour of day, in range 0-23.
`%l`	Hour of day, in range 1-12.
`%m`	Month number, in range 01-12.
`%M`	Minute, in range 00-59.
`%p`	Either `am` or `pm`.
`%S`	Second, in range 00-59.
`%y`	Last two digits of year number.
`%Y`	Year number.
`%Z`	Timezone name (e.g. UTC, CEST, EDT).

The third argument is also optional, and specifies the timezone that the time should be displayed in. As above, this should be specified in the form `Europe/London`, `America/New_York` or `Australia/Perth`, as used by the `tz database`. A complete list of available timezones can be found here: http://en.wikipedia.org/wiki/List_of_tz_database_time_zones. If universal time is used, the timezone may be specified as `UTC`. If no timezone is specified, the default is used as set in the `set timezone` command.

Several functions are provided for converting `date` objects back into various numerical forms of timekeeping and components of calendar dates, which are listed below. Where appropriate, an optional timezone may be specified to obtain a calendar date for a particular location:

toDayOfMonth(< *timezone* >**)**
The toDayOfMonth(< *timezone* >) method returns the day of the month of a date object in the current calendar.

toDayWeekName(< *timezone* >**)**
The toDayWeekName(< *timezone* >) method returns the name of the day of the week of a date object.

toDayWeekNum(< *timezone* >**)**
The toDayWeekNum(< *timezone* >) method returns the day of the week (1–7) of a date object.

toHour(< *timezone* >**)**
The toHour(< *timezone* >) method returns the integer hour component (0–23) of a date object.

toJD()
The toJD() method converts a date object to a numerical Julian date.

toMinute(< *timezone* >**)**
The toMinute(< *timezone* >) method returns the integer minute component (0–59) of a date object.

toMJD()
The toMJD() method converts a date object to a modified Julian date.

toMonthName(< *timezone* >**)**
The toMonthName(< *timezone* >) method returns the name of the month in which a date object falls.

toMonthNum(< *timezone* >**)**
The toMonthNum(< *timezone* >) method returns the number (1–12) of the month in which a date object falls.

toSecond(< *timezone* >**)**
The toSecond(< *timezone* >) method returns the seconds component (0–60) of a date object, including the non-integer component.

toUnix()
The toUnix() method converts a date object to a Unix time.

toYear(< *timezone* >**)**

The toYear(< *timezone* >) method returns the year in which a date object falls in the current calendar.

For example:

```
pyxplot> a = time.fromCalendar(2000,1,1,0,0,0)
pyxplot> time.string(a)
Sat 2000 Jan 1 00:00:00 UTC
pyxplot> time.string(a,"%d %B %Y")
1 January 2000
pyxplot> set calendar muslim
pyxplot> time.string(a,"%d %B %Y")
21 Dhu l-Qa'da 1389
```

4.11.1 Calendars

By default, the `time.fromCalendar` function makes a transition from the old Julian calendar to the new Gregorian calendar at midnight on 14th September 1752 (Gregorian calendar), when Britain and the British Empire switched calendars. Thus, dates between 2nd September and 14th September 1752 are not valid input dates, since they days never occurred in the British calendar.

This behaviour may be changed using the `set calendar` command, which offers a choice of nine different calendars listed in Table 4.8. Most of the these calendars differ only in the date on which the transition is made between the old (Julian) calendar and the new (Gregorian) calendar.

The exceptions are the Hebrew and Islamic calendars, which have entirely different systems of months.

Optionally, the `set calendar` command can be used to set different calendars to use when converting calendar dates into `date` objects, and when converting in the opposite direction. This is useful when converting data from one calendar to another. The syntax used to do this is as follows:

```
set calendar in Julian     # only applies to time.fromCalendar()
set calendar out Gregorian # does not apply to time.fromCalendar()
set calendar in Julian out Gregorian   # change both
show calendar              # show calendars currently being used
```

Example 7: Calculating the date of Leo Tolstoy's birth.

The Russian novelist Leo Tolstoy was born on 28th August 1828 and died on 7th November 1910 in the Russian calendar. What dates do these correspond to in the Western calendar?

```
pyxplot> set calendar in russian out british
pyxplot> birth = time.fromCalendar(1828, 8,28,12,0,0)
pyxplot> death = time.fromCalendar(1910,11, 7,12,0,0)
pyxplot> print birth
Tue 1828 Sep 9 12:00:00 UTC
pyxplot> print death
Sun 1910 Nov 20 12:00:00 UTC
```

Calendar	Description
British	Use the Gregorian calendar from 14th September 1752 (Gregorian), and the Julian calendar prior to 2nd September 1752 (Julian).
French	Use the Gregorian calendar from 20th December 1582 (Gregorian), and the Julian calendar prior to 9th December 1582 (Julian).
Greek	Use the Gregorian calendar from 1st March 1923 (Gregorian), and the Julian calendar prior to 15th February 1923 (Julian).
Gregorian	Use the Gregorian calendar for all dates.
Hebrew	Use the Hebrew (Jewish) calendar.
Islamic	Use the Islamic (Muslim) calendar. Note that the Islamic calendar is undefined prior to 1st Muharram AH 1, corresponding to 18th July AD 622.
Julian	Use the Julian calendar for all dates.
Papal	Use the Gregorian calendar from 15th October 1582 (Gregorian), and the Julian calendar prior to 4th October 1582 (Julian).
Russian	Use the Gregorian calendar from 14th February 1918 (Gregorian), and the Julian calendar prior to 31st January 1918 (Julian).

Table 4.8: The calendars supported by the `set calendar` command, which can be used to convert dates between calendar dates and Julian Day numbers.

4.11.2 Time intervals

The time interval between two date objects can be found by subtracting one from the other. The following example calculates the time interval between Albert Einstein's birth and death. The result is returned as a numerical object with physical dimensions of time:

```
pyxplot> myDate1 = time.fromCalendar(1879,3,14,0,0,0)
pyxplot> myDate2 = time.fromCalendar(1955,4,18,0,0,0)
pyxplot> print myDate2 - myDate1
2401315200 s
pyxplot> print (myDate2 - myDate1) / unit(year)
76.094714
```

The function `time.interval(t1,t2)` has the same effect. The next example calculate the time elapsed between the traditional date for the foundation of Rome by Romulus and Remus in 753 BC and that of the deposition of the last Emperor of the Western Empire in AD 476:

```
pyxplot> x = time.fromCalendar(-752,4,21,12,0,0)
pyxplot> y = time.fromCalendar( 476,9, 4,12,0,0)
pyxplot> print y-x
3.8764483e+10 s
pyxplot> print time.interval(y,x)
3.8764483e+10 s
pyxplot> print (y-x)/unit(year)
1228.3986
```

The function `time.intervalStr()` is similar, but returns a textual representation of the time interval. It takes an optional third parameter which specifies the textual format in which the time interval should be represented. If no format is supplied, then the following verbose format is used:

```
"%Y years %d days %h hours %m minutes and %s seconds"
```

Table 4.10 lists the tokens which are substituted for various parts of the time interval. The following examples demonstrate the use of the function:

```
pyxplot> x = time.fromCalendar(-752,4,21,12,0,0)
pyxplot> y = time.fromCalendar( 476,9, 4,12,0,0)
pyxplot> print time.intervalStr(y,x)
3 hours 14 minutes 8 seconds
pyxplot> print time.intervalStr(y,x,"$%Y^\mathrm{y}%d^\mathrm{d}$")
$-1229^\mathrm{y}-78^\mathrm{d}$
```

Example 8: A plot of the rate of downloads from an Apache webserver.

Token	Substitution value
%%	A literal % sign.
%d	The number of days elapsed, modulo 365.
%D	The number of days elapsed.
%h	The number of hours elapsed, modulo 24.
%H	The number of hours elapsed.
%m	The number of minutes elapsed, modulo 60.
%M	The number of minutes elapsed.
%s	The number of seconds elapsed, modulo 60.
%S	The number of seconds elapsed.
%Y	The number of years elapsed.

Table 4.10: Tokens which are substituted for various components of the time interval by the `time_diff_string` function.

In this example, we use Pyxplot's facilities for handling dates and times to produce a plot of the rate of downloads from an Apache webserver based upon the download log which it stores in the file `/var/log/apache2/access.log`. This file contain a line of the following form for each page or file requested from the webserver:

```
127.0.0.1 - - [14/Jun/2012:16:43:18 +0100] "GET / HTTP/1.1" 200 484 "-"
"Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.19 (KHTML, like Gecko)
Ubuntu/12.04 Chromium/18.0.1025.151 Chrome/18.0.1025.151 Safari/535.19"
```

However, Pyxplot's default input filter for `.log` files (see Section 5.1) manipulates the dates in strings such as these into the form

```
127.0.0.1 - -  [ 14  6  2012 16 43 18 +0100 ]  "GET   HTTP 1.1" 200 484
"-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.19 (KHTML, like Gecko)
Ubuntu/12.04 Chromium/18.0.1025.151 Chrome/18.0.1025.151 Safari/535.19"
```

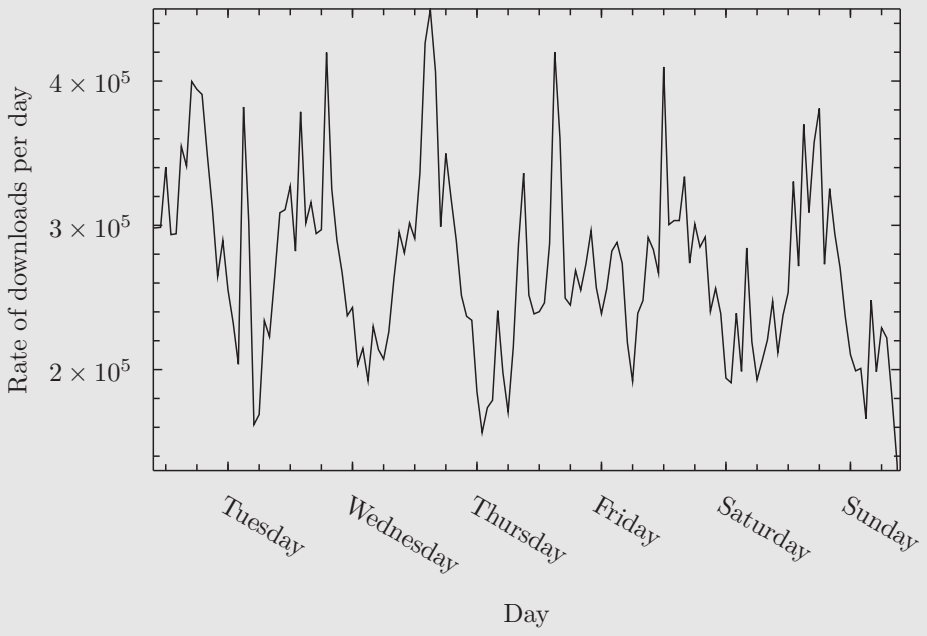
such that the day, month, year, hour, minute and second components of the date are contained in the 5th to 10th white-space-separated columns respectively. In the script below, the `time.fromCalendar()` function and `toUnix()` method are then used to convert these components into Unix times. The `histogram` command (see Section 5.9) is used to sort each of the web accesses recorded in the Apache log file into hour-sized bins. Because this may be a time-consuming process for large log files on busy servers, we use the `tabulate` command (see Section 5.5) to store the data into a temporary data fileon disk before deciding how to plot it:

```
set output 'apache.dat'
histogram f() '/var/log/apache2/access.log' \
using time.fromCalendar($7,$6,$5,$8,$9,$10).toUnix() \
binwidth 1/24
tabulate f(x) with format "%16f %16f"
```

Having stored our histogram in the file `apache.dat`, we now plot the resulting histogram, labelling the horizontal axis with the days of the week. The commands used to achieve this will be introduced in Chapter 8. The major axis ticks along the horizontal axis are placed at daily intervals, and minor axis ticks are placed along the axis every quarter day, i.e. every six hours.

```
set width 10
set xlabel 'Day'
set ylabel 'Rate of downloads per day'
set xtics 0, 86400
set mxtics 0, 21600
set xformat "%s"%(time.fromUnix(x).toDayWeekName()) rot 30
set xrange [1269855360:1270373760]
plot "apachelog.dat" notitle with lines
```

The plot below shows the graph which results on a moderately busy webserver which hosts, among many other sites, the Pyxplot website:



Chapter 5

Working with data

This chapter returns to Pyxplot's commands for acting on data stored in files. Chapter 3 has already introduced the `plot` command, which draws graphs, but there are also commands for tabulating data to new data files, for computing histograms, for interpolating data, and for taking Fourier transforms.

Section 3.8 has already introduced the options which can be used to select data from particular columns or which satisfy particular criteria: `using`, `index`, `every` and `select`. These options are universal to all of Pyxplot's commands which operate on data sets. In all cases, data sets can be read from files, sampled from functions, or specified as a colon-separated list of vectors (see Section 6.5.3).

This chapter begins by describing other common features of these commands, before moving on to describe each command in turn. It leaves the details of the `plot` command, which was introduced in Chapter 3, to be described in full detail in Chapter 8.

5.1 Input filters

By default, Pyxplot expects data files to be in a simple plaintext format which is described in Section 3.8. However, input filters provide a mechanism by which data files in arbitrary formats can be read.

An input filter is specified to act on all data files that match some filename pattern. For example, a filter could be defined to act on all data files called `*.txt` or `*.dat`. The filter itself takes the form of a program which is launched by Pyxplot whenever a matching data file is read. The program is passed the filename of the data file as a command line argument immediately following any arguments specified in the filter's definition. It is then expected to return the data contained in the file to Pyxplot in plaintext format using its `stdout` stream. Any errors which such a program returns to `stderr` are passed to the user as error messages.

Pyxplot has five input filters built-in, as the `show filters` command reveals:

```
set filter "*.fits"   "/usr/local/lib/pyxplot/pyxplot_fitshelper"
set filter "*.gz"     "/bin/gunzip -c"
set filter "*.log"    "/usr/local/lib/pyxplot/pyxplot_timehelper"
set filter "ftp://*"  "/usr/bin/wget -O -"
```

```
set filter "http://*" "/usr/bin/wget -O -"
```

The above set of filters allow Pyxplot to read data from gzipped plaintext data files, from data files available over the web via HTTP or FTP, and data tables in FITS format. A filter is also provided for converting textual dates in log files into numbers representing days, months and years. To add to this list of filters, it is necessary to write a short program or shell script; the simple filters provided in Pyxplot's source code for `.log` and .fits files may provide a useful model.

The filter can then be installed using the syntax

```
set filter <filenameWildcard> <shellCommand>
```

5.2 Reading data from a pipe

Pyxplot usually reads data from files, or samples it from functions. However, it is also possible to read data *piped* into it from other processes if these are directed to Pyxplot's standard input stream, `stdin`. To do this, the magic filename `-` is used:

```
plot '-' with lines
```

This makes it possible to call Pyxplot from within another program and pass it data to plot without ever storing the data on disk. Whilst this facility has great power, it should be used with caution.

It is often very tempting to write programs which both perform calculations and plot the results immediately, but this can make it difficult to replot graphs later. A few months after the event, when the need arises to replot the same data in a different form or in a different style, remembering how to use a sizable program can prove tricky – especially if the person struggling to do so is not you! But a simple data file is quite straightforward to plot time and again.

5.3 Including data within command scripts

It is also possible to embed data directly within Pyxplot scripts, which may be useful when a small number of markers are wanted at particular pre-defined positions on a graph, or when it is desirable to roll a Pyxplot script and the data it takes into a single file for easy storage or transmission. To do this, one uses the magic filename `--` and terminates the data with the string `END`:

```
plot '--' with lines
0 0
1 1
2 0
3 1
END
print "More Pyxplot commands can be placed after END"
```

5.4 Special comment lines in data files

Whilst most comment lines in data files – those lines which begin with a hash character – are ignored by Pyxplot, lines which begin with any of the following four strings are parsed:

```
# Columns:
# ColumnUnits:

# Rows:
# RowUnits:
```

The first pair of special comments affect the behaviour of Pyxplot when plotting `using columns`, while the second pair affect the behaviour of Pyxplot when plotting `using rows` (see Section 3.9.1). Within each pair, the first may be used to tell Pyxplot the names of each of the columns/rows in the data file, while the second may be used to tell Pyxplot the physical units of the values in each of the columns/rows. These special comments may appear multiple times throughout a single data file to indicate changes to the format of the data.

For example, a data file prefixed with the lines

```
# Columns:      Time   Distance
# ColumnUnits:   s      10*m
```

contains two columns of data, the first containing times measured in seconds and the second containing distances measured in tens of metres. Note that because the entries on each of these lines are whitespace-separated, spaces are not allowed in column names or within units such as `10*m`. This data file could be plotted using any of the following forms equivalently

```
plot 'data' using Time:Distance
plot 'data' using $Time:$Distance
plot 'data' using 1:2
```

and the axes of the graph would indicate the units of the data (see Section 8.8.3).

5.5 Tabulating functions and slicing data files

Pyxplot's `tabulate` command is similar to its `plot` command, but instead of plotting a series of data points onto a graph, it writes them to a data file. It can be used to produce text files containing samples of functions, to rearrange/filter the columns in data files, to produce a copy of a data file using different physical units, and so on.

The following example would produce a data file called `gamma.dat` containing a list of values of the gamma function:

```
set output 'gamma.dat'
tabulate [1:5] gamma(x)
```

One way to tabulate multiple functions into a common file is with the `using` modifier, as in the example

```
tabulate [0:2*pi] sin(x):cos(x):tan(x) using 1:2:3:4
```

This tabulates the supplied functions horizontally alongside one another in a series of columns. As many expressions may be supplied to the `using` modifier as columns are wanted.

Alternatively, if a series on functions or data files are listed in a comma-separated list (as is done in the `plot` command to plot multiple datasets), the functions are tabulated one after another in a series of index blocks separated by double linefeeds (see Section 3.8):

```
tabulate [0:2*pi] sin(x), cos(x), tan(x)
```

The `set samples` command can be used to control the number of points that are listed when tabulating functions, in the same way that it controls the number of data points drawn by the `plot` command:

```
set samples 200
```

If the abscissa axis is set to be logarithmic then the functions are evaluated at logarithmically-space points along the axis; otherwise, they are samples at linearly-spaced points.

The `select`, `using` and `every` modifiers operate in the same manner in the `tabulate` command as in the `plot` command. Thus the following example would write out the third, sixth and ninth columns of the data file `input.dat`, but only when the arcsine of the value in the fourth column is positive:

```
set output 'filtered.dat'
tabulate 'input.dat' using 3:6:9 select (asin($4)>0)
```

The numerical display format used for each column of the output file is automatically chosen to preserve accuracy whilst simultaneously being as easily human readable as possible. Thus, columns which contain only integers are displayed as such, and scientific notation is only used in columns which contain very large or very small values.

If desired, however, a custom format may be specified using the `with format` modifier. This can be used both to specify text to appear in between the columns of data, and to specify the format of the data itself using tokens such as `%.5f`, as used by Pyxplot's string substitution operator (`%`; see Section 6.2.1), and the `sprintf` statement of many other programming languages.

For example, to tabulate the values of x^2 to very many significant figures with some additional text, one could use:

```
tabulate x**2 with format "x = %f ; x**2 = %27.20e"
```

This might produce the following output:

```
x = 0.000000 ; x**2 =  0.00000000000000000000e+00
x = 0.833333 ; x**2 =  6.94444444444442421371e-01
x = 1.666667 ; x**2 =  2.77777777777778167589e+00
```

There is flexibility as to how many substitution tokens appear in the format specification. If the number of tokens is fewer than the number of columns of data, then the format is repeated until all the columns have been printed. Thus, the command

```
tabulate x**2 with format "%.3f "
```

might produce the output:

```
0.000 0.000
0.833 0.694
1.667 2.778
```

Note that the space character at the end of the format is important to ensure that there is a gap between the columns.

If formats are supplied for more columns than are present, then the final columns are padded with `nan` (not a number).

The data produced by the `tabulate` command can be sorted in order of any arbitrary metric by supplying an expression after the `sortby` modifier. The data are sorted in order from the lowest value of this expression to the highest.

5.6 Function fitting

The `fit` command can be used to fit arbitrary functional forms to data points read from files. It can be used to produce best-fit lines[1] for datasets, or to determine gradients and other mathematical properties of data by looking at the parameters associated with the best-fitting functional form.

The following simple example fits a straight line to data in a file called `data.dat`:

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

The first line specifies the functional form which is to be used. The coefficients of this function, `a` and `b`, which are to be varied during the fitting process, are listed after the keyword `via` in the `fit` command. The modifiers `index`, `every`, `select` and `using` have the same meanings as in the `plot` command.

When a function of n variables is being fit, at least $n+1$ columns (or rows – see Section 3.9.1) of data 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 arguments to the function, plus finally the value which the output from the function is aiming to match.

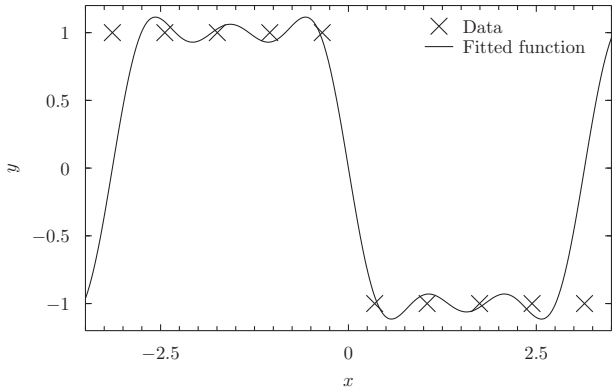
If an additional column is specified, then this is taken to contain the standard error in the value that the output from the function is aiming to match, and can be used to weight the data points which are being used to constrain the fit.

As an example, below we generate a data file containing samples of a square wave using the `tabulate` command and fit the first three terms of a truncated Fourier series to it:

```
set samples 10
set output 'square.dat'
tabulate [-pi:pi] 1-2*heaviside(x)
```

[1] Another way of producing best-fit lines is to use the `interpolate` command; more details are given in Section 5.7

```
f(x) = a1*sin(x) + a3*sin(3*x) + a5*sin(5*x)
fit f() 'square.dat' via a1, a3, a5
set xlabel '$x$' ; set ylabel '$y$'
plot 'square.dat' title 'data' with points pointsize 2, \
     f(x) title 'Fitted function' with lines
```



As the `fit` command works, it displays statistics including the best fit values of each of the fitting parameters, the uncertainties in each of them, and the covariance matrix. These can be useful for analysing the security of the fit achieved, but calculating the uncertainties in the best fit parameters and the covariance matrix can be time consuming, especially when many parameters are being fitted simultaneously. The optional word `withouterrors` can be included immediately before the filename of the input data file to substantially speed up cases where this information is not required.

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() 'data.dat' index 1 using 2:3 via a,b
```

If any of the fitting coefficients are not dimensionless – that is, they have physical units such as meters or seconds – then an initial value with the appropriate units *must* be specified.

A few points are worth noting:

- A series of ranges may be specified after the `fit` command, using the same syntax as in the `plot` command, as described in Section 3.14. If ranges are specified then only data points 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:

  ```
  fit [0:10] f() 'data.dat' via a
  ```

- As with all numerical fitting procedures, the `fit` command comes with caveats. It uses a generic fitting algorithm, and may not work well with

poorly behaved or ill-constrained problems. It works best when all of the values it is attempting to fit are of order unity. For example, in a problem where a was of order 10^{10}, the following might fail:

```
f(x) = a*x
fit f() 'data.dat' 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() 'data.dat' via a
```

- For those interested in the mathematical details, the workings of the `fit` command are discussed in more detail in Appendix C.

5.7 Datafile interpolation

The `interpolate` command can be used to generate a special function within Pyxplot's mathematical environment which interpolates a set of data points supplied from a data file. As with other commands, data can also be supplied from functions, or from a colon-separated list of vectors (see Section 6.5.3). Either one- or two-dimensional interpolation is possible. Two-dimensional interpolation is described in the next section.

In the case of one-dimensional interpolation, various different types of interpolation are supported: linear interpolation, power law interpolation, polynomial interpolation, cubic spline interpolation and akima spline interpolation. Stepwise interpolation returns the value of the datapoint nearest to the requested point in argument space. The use of polynomial interpolation with large datasets is strongly discouraged, as polynomial fits tend to show severe oscillations between data points.

Except in the case of stepwise interpolation, extrapolation is not permitted; if an attempt is made to evaluate an interpolated function beyond the limits of the data points which it interpolates, Pyxplot returns an error or value of not-a-number. This behaviour can be configured using the `set numeric errors quiet` command (see Section 4.4).

The `interpolate` command has similar syntax to the `fit` command:

```
interpolate ( akima | linear | loglinear | polynomial |
              spline | stepwise |
              2d [( bmp_r | bmp_g | bmp_b )] )
            [<range specification>] <function name>"()"
            '<filename>'
            [ every <expression> {:<expression} ]
            [ index <value> ]
            [ select <expression> ]
            [ using <expression> {:<expression} ]
```

A very common application of the `interpolate` command is to perform arithmetic functions such as addition or subtraction on datasets which are not

sampled at the same abscissa values. The following example would plot the difference between two such datasets:

```
interpolate linear f() 'data1.dat'
interpolate linear g() 'data2.dat'
plot [min:max] f(x)-g(x)
```

Note that it is advisable to supply a range to the `plot` command in this example: because the two datasets have been turned into continuous functions, the `plot` command has to guess a range over which to plot them, unless one is explicitly supplied.

The `spline` command is an alias for `interpolate spline`; the following two statements are equivalent:

```
spline f() 'data1.dat'
interpolate spline f() 'data1.dat'
```

Example 9: A demonstration of the `linear`, `spline` and `akima` modes of interpolation.

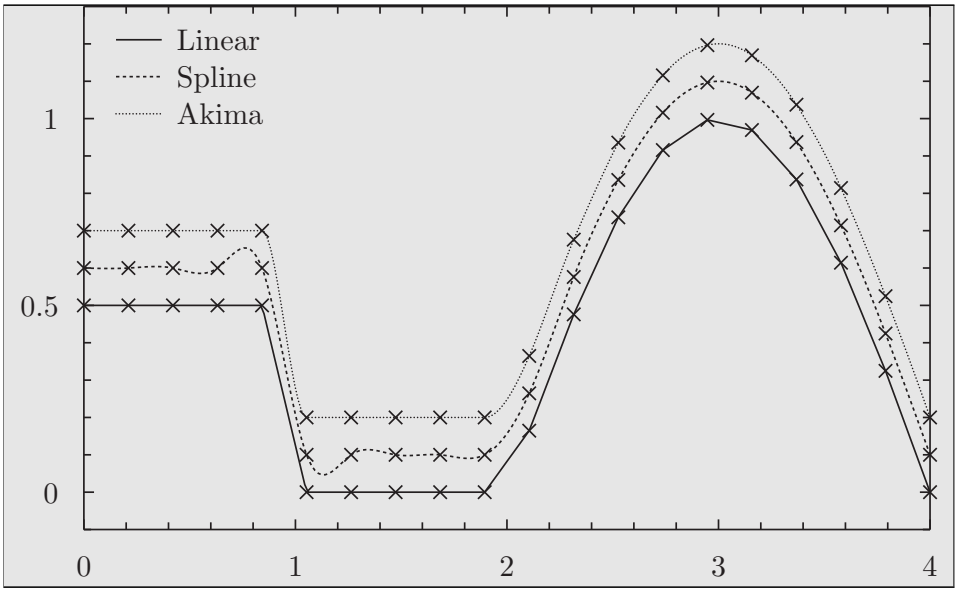
In this example, we demonstrate the `linear`, `spline` and `akima` modes of interpolation using an example data file with non-smooth data generated using the `tabulate` command (see Section 5.5):

```
f(x) = 0
f(x)[0:1] = 0.5
f(x)[2:4] = cos((x-3)*pi/2)
set samples 20
tabulate [0:4] f(x)
```

Having set three functions to interpolate these non-smooth data in different ways, we plot them with a vertical offset of 0.1 between them for clarity. The interpolated data fileis plotted with points three times to show where each of the interpolation functions is pinned.

```
interpolate linear f_linear() "interpolation.dat"
interpolate spline f_spline() "interpolation.dat"
interpolate akima  f_akima () "interpolation.dat"

set key top left
plot [0:4][-0.1:1.3] \
"interpolation.dat" using 1:($2+0.0) notitle with points pt 1, \
f_linear(x)+0.0 title "Linear", \
"interpolation.dat" using 1:($2+0.1) notitle with points pt 1, \
f_spline(x)+0.1 title "Spline", \
"interpolation.dat" using 1:($2+0.2) notitle with points pt 1, \
f_akima (x)+0.2 title "Akima"
```

The resulting plot is shown below:



5.7.1 Two-dimensional interpolation

In the case of two-dimensional interpolation, the type of interpolation to be used is set using the `interpolate` modifier to the `set samples` command, and may be changed at any time after the interpolation function has been created. The options available are nearest neighbor interpolation – which is the two-dimensional equivalent of stepwise interpolation, inverse square interpolation – which returns a weighted average of the supplied data points, using the inverse squares of their distances from the requested point in argument space as weights, and Monaghan Lattanzio interpolation, which uses the weighting function (Monaghan & Lattanzio 1985)

$$\begin{aligned} w(x) &= 1 - {}^3\!/_2 v^2 + {}^3\!/_4 v^3 \quad \text{for } 0 \le v \le 1 \\ &= {}^1\!/_4 (2 - v)^3 \quad \text{for } 1 \le v \le 2 \end{aligned}$$

where $v = r/h$ for $h = \sqrt{A/n}$, A is the product $(x_{\max} - x_{\min})(y_{\max} - y_{\min})$ and n is the number of input datapoints. These are selected as follows:

```
set samples interpolate nearestNeighbor
set samples interpolate inverseSquare
set samples interpolate monaghanLattanzio
```

The following example creates a function `quadrapole(x,y)` which interpolates a quadrapole:

```
set samples interpolate inverseSquare
interpolate 2d quadrapole() '--'
-1 -1  1
-1  1 -1
 1 -1 -1
 1  1  1
END
```

Finally, data can be imported from graphical images in bitmap (`.bmp`) format to produce a function of two arguments returning a value in the range $0 \to 1$ which represents the data in one of the image's three color channels. The two arguments are the horizontal and vertical position within the bitmap image, as measured in pixels. This is done using syntax of the form:

```
interpolate 2d bmp_b blue() 'myImg.bmp'
```

5.8 Fourier transforms

The `fft` and `ifft` commands take Fourier transforms and inverse Fourier transforms respectively of data. As with other commands, data can be supplied from a data file, from functions, or from a colon-separated list of vectors (see Section 6.5.3). In each case, a regular grid of abscissa values must be specified on which to take the discrete Fourier transform, which can extend over an arbitrary number of dimensions. The following example demonstrates the syntax of these commands as applied to a two-dimensional top-hat function:

```
step(x,y) = tophat(x,0.2) * tophat(y,0.4)
fft  [  0: 1:0.01][  0: 1:0.01] f() of step()
ifft [-50:49:1   ][-50:49:1   ] g() of f()
```

In the `fft` command above, $N_x = 100$ equally-spaced samples of the function `step`(x, y) are taken between limits of $x_{\min} = 0$ and $x_{\max} = 1$ for each of $N_y = 100$ equally-spaced values of y on an identical raster, giving a total of 10^4 samples. These are converted into a rectangular grid of 10^4 samples of the Fourier transform `f`(ω_x, ω_y) at

$$\omega_x = \frac{j}{\Delta x} \quad \text{for} \quad -\frac{N_x}{2} \le j < \frac{N_x}{2} \quad \left(\text{equivalently, for} -\frac{N_x}{2\Delta x} \le \omega_x < \frac{N_x}{2\Delta x}\right),$$
$$\omega_y = \frac{k}{\Delta y} \quad \text{for} \quad -\frac{N_y}{2} \le k < \frac{N_y}{2} \quad \left(\text{equivalently, for} -\frac{N_y}{2\Delta y} \le \omega_y < \frac{N_y}{2\Delta y}\right).$$

where $\Delta x = x_{\max} - x_{\min}$ and Δy is analogously defined. These samples are interpolated stepwise, such that an evaluation of the function `f`(ω_x, ω_y) for general inputs yields the nearest sample, or zero outside the rectangular grid where samples are available. In general, even the Fourier transforms of real functions are complex, and their evaluation when complex arithmetic is not enabled (see Section 4.5) is likely to fail. For this reason, a warning is issued if complex arithmetic is disabled when a Fourier transform function is evaluated.

In the example above, we go on to convert this set of samples back into the function with which we started by instructing the `ifft` command to take $N_x = 100$ equally-spaced samples along the ω_x-axis between $\omega_{x,\min} = -N_x/2\Delta x$ and $\omega_{x,\max} = (N_x - 1)/2\Delta x$, with similar sampling along the ω_y-axis.

Taking the simpler example of a one-dimensional Fourier transform for clarity, as might be calculated by the instructions

```
step(x) = tophat(x,0.2)
fft  [  0: 1:0.01] f() of step()
```

the `fft` and `ifft` commands compute, respectively, discrete sets of samples F_m and I_n of the functions $F(\omega_x)$ and $I(x)$, which are given by

$$F_j = \sum_{m=0}^{N-1} I_m e^{-2\pi i j m/N}\, \delta x, \text{ for } -\frac{N}{2} \le j < \frac{N}{2},$$

and

$$I_j = \sum_{m=0}^{N-1} F_m e^{2\pi i j m/N}\, \delta\omega_x, \text{ for } -\frac{N}{2} \le j < \frac{N}{2},$$

where:

$I(x)$	=	Function being Fourier transformed.
$F(\omega_x)$	=	Fourier transform of $I()$.
N	=	The number of values sampled along the abscissa axis.
δx	=	Spacing of values sampled along the abscissa axis.
$\delta\omega_x$	=	Spacing of abscissa values sampled along the ω_x axis.
i	=	$\sqrt{-1}$.

It may be shown in the limit that δx becomes small – i.e. when the number of samples taken becomes very large – that these sums approximate the integrals

$$F(\omega_x) = \int I(x) e^{-2\pi i x \omega_x}\, \mathrm{d}x, \tag{5.1}$$

and

$$I(x) = \int F(\omega_x) e^{2\pi i x \omega_x}\, \mathrm{d}\omega_x. \tag{5.2}$$

Fourier transforms may also be taken of data stored in data files using syntax of the form

```
fft [-10:10:0.1] f() of 'datafile.dat'
```

In such cases, the data read from the data file for an N-dimensional FFT must be arranged in $N+1$ columns[2], with the first N containing the abscissa values for each of the N dimensions, and the final column containing the data to be Fourier transformed. The abscissa values must strictly match those in the raster specified in the `fft` or `ifft` command, and must be arranged strictly in row-major order.

Example 10: The Fourier transform of a top-hat function.

[2]The `using`, `every`, `index` and `select` modifiers can be used to arrange it into this form.

It is straightforward to show that the Fourier transform of a top-hat function of unit width is the function $\mathrm{sinc}(x' = \pi x) = \sin(x')/x'$. If

$$f(x) = \begin{cases} 1 & |x| \leq 1/2 \\ 0 & |x| > 1/2 \end{cases},$$

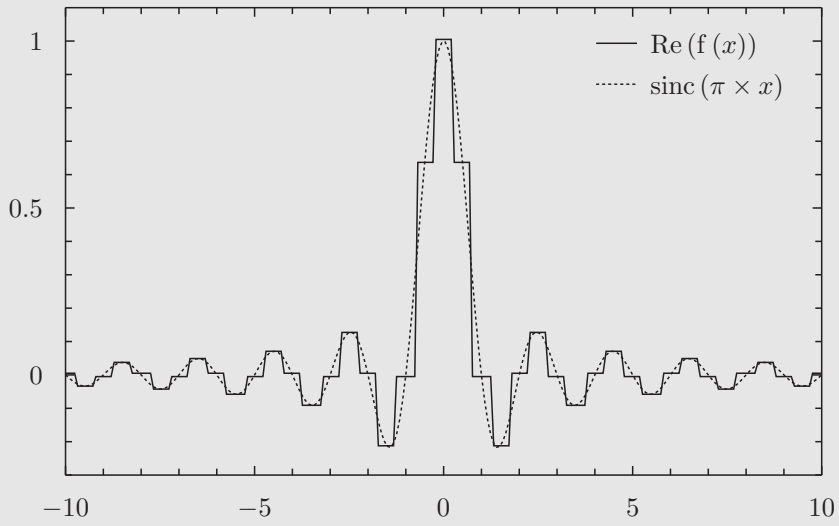
the Fourier transform $F(\omega)$ of $f(x)$ is

$$\begin{aligned} F(\omega) &= \int_0^\infty f(x) \exp(-2\pi i x\omega)\, \mathrm{d}x = \int_{-1/2}^{1/2} \exp(-2\pi i x\omega)\, \mathrm{d}x \\ &= \frac{1}{2\pi\omega} \left[\exp(\pi i\omega) - \exp(-\pi i\omega)\right] = \frac{\sin(\pi\omega)}{\pi\omega} = \mathrm{sinc}(\pi\omega). \end{aligned}$$

In this example, we demonstrate this numerically by taking the Fourier transform of such a step function, and comparing the result against the function `sinc(x)` which is pre-defined within Pyxplot:

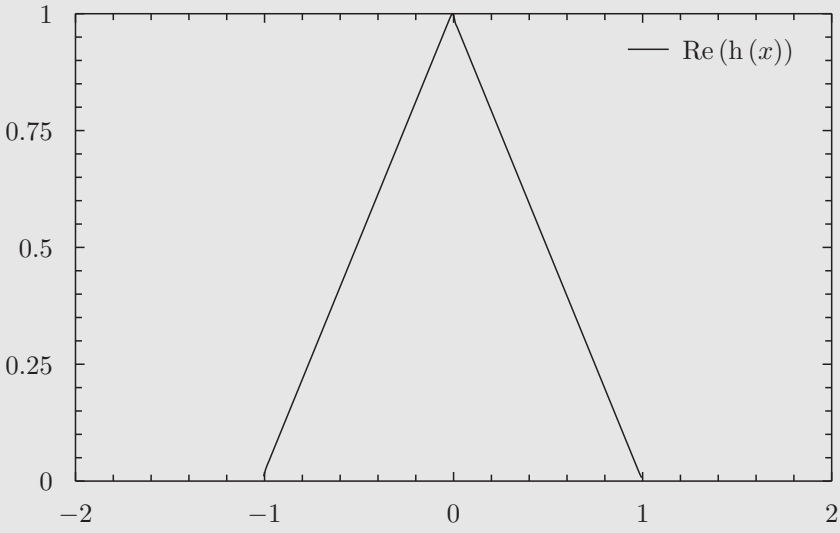
```
set numerics complex
step(x) = tophat(x,0.5)
fft [-1:1:0.01] f() of step()
plot [-10:10] Re(f(x)), sinc(pi*x)
```

Note that the function `Re(x)` is needed in the `plot` statement here, since although the Fourier transform of a symmetric function is in theory real, in practice any numerical Fourier transform will yield a small imaginary component at the level of the accuracy of the numerical method used. Although the calculated numerical Fourier transform is defined throughout the range $-50 \leq x < 50$, discretised with steps of size $\Delta x = 0.5$, we only plot the central region in order to show clearly the stepping of the function:



In the following steps, we take the square of the function $\mathrm{sinc}(\pi x)$ just calculated, and then plot the numerical inverse Fourier transform of the result:

```
g(x) = f(x)**2
ifft [-50:49.5:0.5] h(x) of g(x)
plot [-2:2] Re(h(x))
```



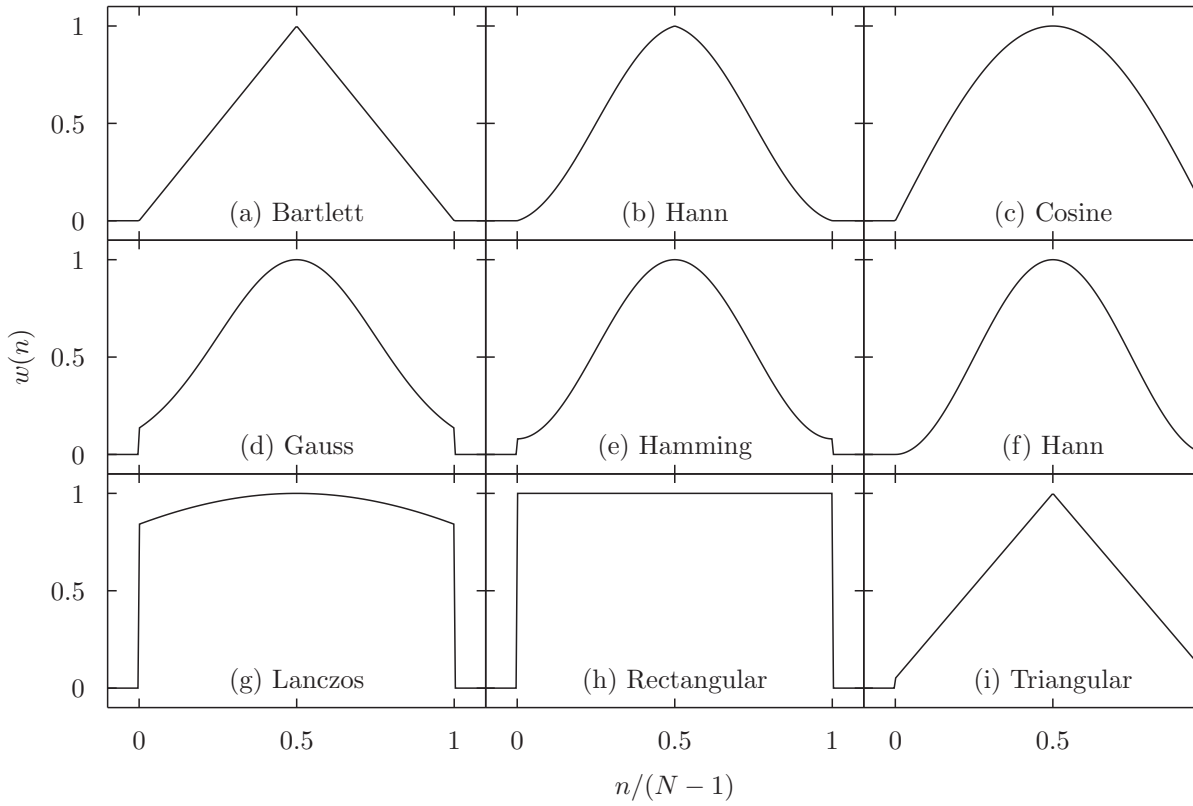


Figure 5.1: Window functions available in the `fft` and `ifft` commands.

As can be seen, the result is a triangle function. This is the result which would be expected from the convolution theorem, which states that when the Fourier transforms of two functions are multiplied together and then inverse transformed, the result is the convolution of the two original functions. The convolution of a top-hat function with itself is, indeed, a triangle function.

5.8.1 Window functions

A range of commonly-used window functions may automatically be applied to data as it is read into the `fft` and `ifft` commands; these are listed together with their algebraic forms in Table 5.3 and shown in Figure 5.1. In each case, the window functions are given for sample number n, which ranges between 0 and N_x. The window functions may be invoked using the following syntax:

```
fft [...] <out>() of <in>() window <window_name>
```

Where multi-dimensional FFTs are performed, window functions are applied to each dimension in turn. Other arbitrary window functions may be implemented by pre-multiplying data before entry to the `fft` and `ifft` commands.

Window Name	**Algebraic Definition**
Bartlett	$w(n) = \left(\frac{2}{N_x - 1}\right)\left(\frac{N_x - 1}{2} - \left\lvert n - \frac{N_x - 1}{2}\right\rvert\right)$
BartlettHann	$w(n) = a_0 - a_1\left\lvert\frac{n}{N_x - 1} - \frac{1}{2}\right\rvert - a_2 \cos\left(\frac{2\pi n}{N_x - 1}\right)$, for $a_0 = 0.62,\ a_1 = 0.48,\ a_2 = 0.38.$
Cosine	$w(n) = \cos\left(\frac{\pi n}{N_x - 1} - \frac{\pi}{2}\right)$
Gauss	$w(n) = \exp\left\{-\frac{1}{2}\left[\frac{n - (N_x - 1)/2}{\sigma(N_x - 1)/2}\right]^2\right\}$, for $\sigma = 0.5$
Hamming	$w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N_x - 1}\right)$
Hann	$w(n) = 0.5\left[1 - \cos\left(\frac{2\pi n}{N_x - 1}\right)\right]$
Lanczos	$w(n) = \mathrm{sinc}\left(\frac{2n}{N_x - 1} - 1\right)$
Rectangular	$w(n) = 1$
Triangular	$w(n) = \left(\frac{2}{N_x}\right)\left(\frac{N_x}{2} - \left\lvert n - \frac{N_x - 1}{2}\right\rvert\right)$

Table 5.3: Window functions available in the `fft` and `ifft` commands.

5.9 Histograms

The `histogram` command takes a single column of data and produces a function that represents the frequency distribution of the supplied data values. The output function consists of a series of discrete intervals which we term *bins*. Within each interval the output function has a constant value, determined such that the area under each interval – i.e. the integral of the function over each interval – is equal to the number of datapoints found within that interval. The following simple example

```
histogram f() 'input.dat'
```

produces a frequency distribution of the data values found in the first column of the file `input.dat`, which it stores in the function $f(x)$. The value of this function at any given point is equal to the number of items in the bin at that point, divided by the width of the bins used. If the input datapoints are not dimensionless then the output frequency distribution adopts appropriate units, thus a histogram of data with units of length has units of one over length.

The number and arrangement of bins used by the `histogram` command can be controlled by means of various modifiers. The `binwidth` modifier sets the width of the bins used. The `binorigin` modifier controls where their boundaries lie; the `histogram` command selects a system of bins which, if extended to infinity in both directions, would put a bin boundary at the value specified in the `binorigin` modifier. Thus, if `binorigin 0.1` were specified, together with a bin width of 20, bin boundaries might lie at 20.1, 40.1, 60.1, and so on. Alternatively global defaults for the bin width and the bin origin can be specified using the `set binwidth` and `set binorigin` commands respectively. The example

```
histogram h() 'input.dat' binorigin 0.5 binwidth 2
```

would bin data into bins between 0.5 and 2.5, between 2.5 and 4.5, and so forth.

Alternatively the set of bins to be used can be controlled more precisely using the `bins` modifier, which allows an arbitrary set of bin boundaries to be specified. The example

```
histogram g() 'input.dat' bins (1, 2, 4)
```

would bin the data into two bins, $x = 1 \to 2$ and $x = 2 \to 4$.

A range can be supplied immediately following the `histogram` command, using the same syntax as in the `plot` and `fit` commands; if such a range is supplied, only points that fall within that range will be binned. In the same way as in the `plot` command, the `index`, `every`, `using` and `select` modifiers can be used to specify which subsets of a data file should be used.

Two points about the `histogram` command are worthy of note. First, although histograms are similar to bar charts, they are not the same. A bar chart conventionally has the height of each bar equal to the number of points that it represents, whereas a histogram is a continuous function in which the area underneath each interval is equal to the number of points within it. Thus, to produce a bar chart using the `histogram` command, the end result should be multiplied by the bin width used.

Second, if the function produced by the `histogram` command is plotted using the `plot` command, samples are automatically taken not at evenly spaced intervals along the abscissa axis, but at the centres of each bin. If the `boxes` plot style is used, the box boundaries are conveniently drawn to coincide with the bins into which the data were sorted.

5.10 Random data generation

Pyxplot has functions for generating random numbers from a variety of common probability distributions. These functions are in the `random` module:

- `random.random()` – returns a random real number between 0 and 1.
- `random.binomial(`p, n`)` – returns a random sample from a binomial distribution with n independent trials and a success probability p.
- `random.chisq(`ν`)` – returns a random sample from a χ-squared distribution with ν degrees of freedom.
- `random.gaussian(`σ`)` – returns a random sample from a Gaussian (normal) distribution of standard deviation σ and centred on zero.
- `random.lognormal(`ζ, σ`)` – returns a random sample from the log normal distribution centred on ζ, and of width σ.
- `random.poisson(`n`)` – returns a random integer from a Poisson distribution with mean n.
- `random.tdist(`ν`)` – returns a random sample from a t-distribution with ν degrees of freedom.

These functions all rely on a common underlying random number generator[3], whose seed may be set using the `set seed` command, which should be followed by any integer. The sequence of random samples generated is always the same after setting any particular seed.

When Pyxplot starts, the seed is implicitly set to zero. **This means that Pyxplot always produces the same series of random numbers when restarted.** This series can be reproduced by typing:

```
set seed 0
```

For applications where this repeatability is undesirable, the following command may help, using the system clock as a random seed:

```
set seed time.now().toUnix()
```

This gives a different sequence of random numbers each second. However, the user is advised to consider carefully whether this is sufficient for the particular application being implemented.

[3]The gsl library's default random number generator, `gsl_rng_default` is used. As of version 1.15, this maps to `gsl_rng_mt19937` with a default seed of zero. The various probability distributions above are sampled using the functions `gsl_ran_binomial` and similar.

Example 11: Using random numbers to estimate the value of π.

Pyxplot's functions for generating random numbers are most commonly used for adding noise to artificially-generated data. In this example, however, we use them to implement a rather inefficient algorithm for estimating the value of the mathematical constant π. The algorithm works by spreading randomly-placed samples in the square $\{-1 < x < 1;\ -1 < y < 1\}$. The number of these which lie within the circle of unit radius about the origin are then counted. Since the square has an area of $4\,\mathrm{unit}^2$ and the circle an area of $\pi\,\mathrm{unit}^2$, the fraction of the points which lie within the unit circle equals the ratio of these two areas: $\pi/4$.

The following script performs this calculation using $N = 5000$ randomly placed samples. Firstly, the positions of the random samples are generated using the `random()` function, and written to a file called `random.dat` using the `tabulate` command. Then, the `foreach datum` command – which will be introduced in Section 7.4 – is used to loop over these, counting how many lie within the unit circle.

```
Nsamples = 500

rand() = random.random()

set samp Nsamples
set output "pi_estimation.dat"
tabulate 1-2*rand():1-2*rand() using 0:2:3

n=0
foreach datum i,j in "pi_estimation.dat" u 2:3
{
n = n + (hypot(i,j)<1)
}
print "pi=%s"%(n / Nsamples * 4)
```

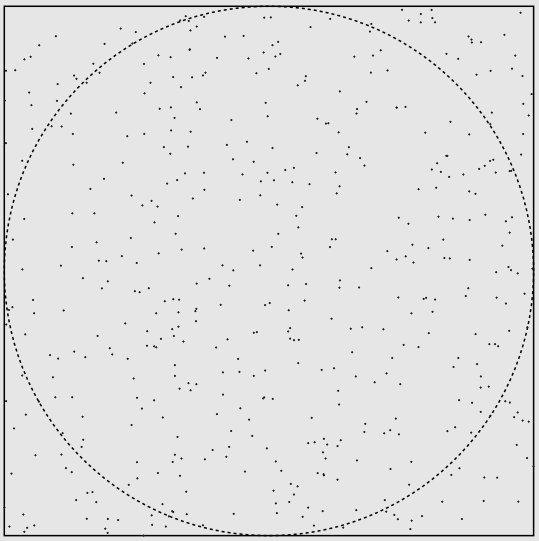
On the author's machine, this script returns a value of 3.1352 when executed using the random samples which are returned immediately after starting Pyxplot. This method of estimating π is well modelled as a Poisson process, and the uncertainty in this result can be estimated from the Poisson distribution to be $1/\sqrt{N}$. In this case, the uncertainty is 0.01, in close agreement with the deviation of the returned value of 3.1352 from more accurate measures of π.

With a little modification, this script can be adapted to produce a diagram of the datapoints used in its calculation. Below is a modified version of the second half of the script, which loops over the data points stored in the data file `random.dat`. It uses Pyxplot's vector graphics commands, which will be introduced in Chapter 10, to produce such a diagram:

```
set multiplot ; set nodisplay
```

```
# Draw a unit circle and a unit square
title = "ex_pi_estimation" ; load "fig_init.ppl"
box from -width/2,-width/2 to width/2,width/2
circle at 0,0 radius width/2 with lt 2

# Now plot the positions of these random data points andi
# count how many lie within a unit circle
n=0
foreach datum i,j in "pi_estimation.dat" using 2:3
{
point at width/2*i , width/2*j with ps 0.1
n = n + (hypot(i,j)<1)
}
set display ; refresh
print "pi=%.4f"%(n / Nsamples * 4)
```

The graphical output from this script is shown below. The number of datapoints has been reduced to `Nsamples`= 500 for clarity:



Chapter 6

Programming: Pyxplot's data types

This chapter describes Pyxplot's built-in object types, which include lists, dictionaries, vectors, matrices and file handles.

All objects in Pyxplot, including numbers, have *methods*, which act on or return information about the object. Some methods are common to all objects. For example, they all have a method `str()` which returns a string representation of the object (as used by the `print` command). All objects also have a method called `methods()`, which returns a list of the names of all of methods of that object. These methods can be called as follows (we don't show the output, as it is long):

```
print pi.str()
print "My son, it's a wisp of fog.".methods()
```

Methods are like functions in that printing them returns brief documentation about them, as we demonstrate below on two methods of string objects:

```
pyxplot> print "Father, do you not see the Elfking?".upper
upper() converts a string to uppercase.
pyxplot> print "Father, do you not see the Elfking?".upper()
FATHER, DO YOU NOT SEE THE ELFKING?
pyxplot> print "Father, do you not see the Elfking?".methods
methods() returns a list of the methods of an object.
```

The following sections describe each of Pyxplot's types in turn, and the methods that can be applied to each of them. A comprehensive list of all of Pyxplot's object types can be found in Chapter 13, which also lists the methods available in each object type.

6.1 Instantiating objects

A list of all of Pyxplot's built-in object types can be found in the `types` module, which contains the *object prototypes* for each type. Its contents are as follows:

```
pyxplot> print types
module {
boolean       : <data type: boolean>
color         : <data type: color>
date          : <data type: date>
dictionary    : <data type: dictionary>
exception     : <data type: exception>
fileHandle    : <data type: fileHandle>
function      : <data type: function>
instance      : <data type: instance>
list          : <data type: list>
matrix        : <data type: matrix>
module        : <data type: module>
null          : <data type: null>
number        : <data type: number>
string        : <data type: string>
type          : <data type: type>
vector        : <data type: vector>
}
```

These object prototypes can be called like functions to produce an instance of each data type. Each prototype can take various different kinds of argument; for example, the `number` prototype can take a `number`, `boolean` or a `string` from which to create a number. For example:

```
pyxplot> types.number()
0
pyxplot> types.number(true)
1
pyxplot> types.number(27)
27
pyxplot> types.number("1.2e39")
1.2e+39
```

Full documentation of the types of inputs supported by each prototype are listed in the Reference Manual, in Section 12.0.11.

In many cases there are much more succinct ways of creating objects of each type. For example, lists can be creating by enclosing a comma-separated list of elements in square brackets:

```
pyxplot> print [10,colors.green,"bottles"]
[10, cmyk(1,0,1,0), "bottles"]
```

Dictionaries can be can be creating by enclosing key–value pairs in curly brackets:

```
pyxplot> s = {"name":"Sophie", "nationality":"British"}
pyxplot> s["hometown"] = "Lode"
pyxplot> s["birthYear"] = 1957
```

Escape sequence	Description
\?	Question mark
\'	Apostrophe
\"	Double quote
\\	Literal backslash
\a	Bell character
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Table 6.1: A complete list of Pyxplot's string escape sequences. These are a subset of those available in C.

6.2 Strings

Strings can be enclosed either in single (') or double (") quotes. Strings may also be enclosed by three quote characters in a row: either ''' or """. Special care needs to be taken when using apostrophes or quotes in single-quote delimited strings, as these characters may be misinterpreted as string delimiters, as in the example:

✗
```
'Robert's data'
```

This easiest way to avoid such problems is to use three quotes:

✔
```
'''Robert's data'''
```

Special characters such as tabs and newlines can be inserted into strings using escape codes such as \t and \n; see Table 6.1 for a list of these. The following string is split over three lines:

```
pyxplot> print e'the moon,\nthe moon,\nThey danced by the light of the moon.'
the moon,
the moon,
They danced by the light of the moon.
```

Sometimes these escape codes can be rather annoying, especially when entering latex control codes, which all begin with backslash characters. Rather than having to escape every backslash, it is generally easier to prefix the string with the character r, which turns off all escape codes:

```
pyxplot> print r'''I escaped the quote by typing \'.'''
I escaped the quote by typing \'.
```

Once defined, a string variable can be used anywhere in Pyxplot where a quoted string could have been used, for example in the `set title` command:

```
plotname = "Insert title here"
set title plotname
```

Strings can be concatenated together using the `+` operator:

```
pyxplot> print "pi = " + pi.str()
pi = 3.1415927
```

6.2.1 The string substitution operator

Most string manipulations are performed using the string substitution operator, `%`, which performs a similar role to the `sprintf` statement in C.

This operator should be preceded by a format string, such as `'x=%f'`, in which tokens such as `%f` mark places where numbers and strings should be substituted. The substitution operator is followed by a bracketed list of the quantities which should be substituted in place of these tokens. This behaviour is similar to that of the Python programming language's % operator[1]

For example, to concatenate the two strings contained in the variables `a` and `b` into a single string variable `c`, one would issue the command:

```
c = "%s%s"%(a,b)
```

One application of this operator might be to label plots with the title of the data file being plotted, as in the following example:

```
filename="data_file.dat"
title=r"A plot of the data in {\tt %s}."%filename
set title title
plot filename
```

The syntax of the substitution tokens placed in the format string is similar to that used by many other languages (including C and Python). All substitution tokens begin with a `%` character, after which there may be placed, in order:

1. An optional minus sign, to specify that the substituted item should be left-justified.

2. An optional integer specifying the minimum character width of the substituted item, or a `*` (see below).

3. An optional decimal point/period (`.`) separator.

4. An optional integer, or a `*` (see below), specifying either (a) the maximum number of characters to be printed from a string, or (b) the number of decimal places of a floating-point number to be displayed, or (c) the minimum number of digits of an integer to be displayed, padded to the left with zeros.

5. A conversion character.

Character	Substitutes
`d`, `i`	An integer value.
`e`, `E`	A floating-point value in scientific notation using either the character `e` or `E` to indicate exponentiation.
`f`	A floating-point value without the use of scientific notation.
`g`, `G`	A floating-point value, either using scientific notation, if the exponent is greater than the precision or less than -4, otherwise without the use of scientific notation.
`o`	An integer value in octal (base 8).
`s`, `S`, `c`	A string, if a string is provided, or a numerical quantity, with units, if such is provided.
`x`, `X`	An integer value in hexadecimal (base 16).
`%`	A literal `%` sign.

Table 6.2: The conversion characters recognised by the string substitution operator, `%`.

The conversion character is a single character which specifies what kind of substitution should take place. Its possible values are listed in Table 6.2.

Where the character `*` is specified for either the character width or the precision of the substitution token, an integer is read from the list of items to be substituted, as happens in C's `printf` command:

```
pyxplot> print "%.*f"%(3,pi)
3.142
pyxplot> print "%.*f"%(6,pi)
3.141593
```

6.2.2 Converting strings to numbers

Strings which contain numerical data can be converted to numbers by passing them to the object `types.number()`, as in the examples:

```
pyxplot> print types.number("23")
23
pyxplot> print types.number("1610 1643 1715 1774".split()[2])
1715
pyxplot> print types.number("978-0230200951"[4:])
230200951
```

It is an error to try to convert a string to a number if it does not contain a correctly-formatted number:

✗ `types.number("this is not a number")`

[1] As in Python, the brackets are optional when only one item is being substituted. For example, `'%d'%2` is equivalent to `'%d'%(2)`.

6.2.3 Slicing strings

Segments of strings can be cut out by using square brackets to slice the string:

```
pyxplot> poem = "On the last sabbath day of 1879\n"
pyxplot> poem+= "Which shall be remembered for a very long time."
pyxplot> print poem[10]
t
pyxplot> print poem[10:]
t sabbath day of 1879\nWhich shall be remembered for a very long time.
pyxplot> print poem[:10]
On the las
pyxplot> print poem[5:10]
e las
pyxplot> print poem[-10:]
long time.
```

If a single number is placed in the square brackets, a single character is taken out of the string. If two colon-separated numbers are specified, `[x:y]`, then the substring from character position `x` up to but not including `y` is returned. If either `x` or `y` are omitted, then the start or end of the string is used respectively. If either number of negative, then it counts from the end of the string, -1 being the last character in the string.

6.2.4 String methods

Strings have many methods for performing simple string manipulations. Here we list their names using the `foreach` command, which will be introduced in the next chapter:

```
pyxplot> foreach m in "".methods() { print m ; }
append
beginsWith
class
contents
data
endsWith
find
findAll
isalnum
isalpha
isdigit
len
lower
lstrip
methods
rstrip
split
splitOn
str
strip
```

```
type
upper
```

Full documentation of them can be found in Section 13.15. As in Python, the `strip()` method removes whitespace characters from the beginning and end of strings, and the `split()` method splits a string up into whitespace-separated words. The `splitOn(x)` method splits a string on all occurrences of the substring `x`. The following examples demonstrate the use of some of them:

```
pyxplot> x="It was the best of times, it was the worst of times"
pyxplot> print x.len()
51
pyxplot> print x.split()[0:5]
["It", "was", "the", "best", "of"]
pyxplot> print x.splitOn(",")
["It was the best of times", " it was the worst of times"]
pyxplot> print x.find("worst")
37
pyxplot> print x[0:24]
It was the best of times
pyxplot> print x[-25:]
it was the worst of times
pyxplot> print x.upper()
IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES
```

```
pyxplot> x="    BEAUTIFUL new railway bridge of the Silvery Tay,"
pyxplot> print x.lstrip()
BEAUTIFUL new railway bridge of the Silvery Tay,
```

6.2.5 Regular expressions

String variables can be modified using the search-and-replace string operator[2], =∼, which takes a regular expression with a syntax similar to that expected by the shell command `sed` and applies it to the specified string variable.[3] In the following example, the first instance of the letter `s` in the string variable `twister` is replaced with the letters `th`:

```
pyxplot> twister="seven silver soda syphons"
pyxplot> twister =∼ s/s/th/
pyxplot> print twister
theven silver soda syphons
```

Note that only the `s` (substitute) command of `sed` is implemented in Pyxplot. Any character can be used in place of the `/` characters in the above example, for example:

[2]Programmers with experience of `perl` will recognise this syntax.

[3]Regular expression syntax is a massive subject, and is beyond the scope of this manual. The official GNU documentation for the `sed` command is heavy reading, but there are many more accessible tutorials on the web.

`g`	Replace *all* matches of the pattern; by default, only the first match is replaced.
`i`	Perform case-insensitive matching, such that expressions like `[A-Z]` will match lowercase letters, too.
`l`	Make `\w`, `\W`, `\b`, `\B`, `\s` and `\S` dependent on the current locale.
`m`	When specified, the pattern character `^` matches the beginning of the string and the beginning of each line immediately following each newline. The pattern character `$` matches at the end of the string and the end of each line immediately preceding each newline. By default, `^` matches only the beginning of the string, and `$` only the end of the string and immediately before the newline, if present, at the end of the string.
`s`	Make the `.` special character match any character at all, including a newline; without this flag, `.` will match anything except a newline.
`u`	Make `\w`, `\W`, `\b`, `\B`, `\s` and `\S` dependent on the Unicode character properties database.
`x`	This flag allows the user to write regular expressions that look nicer. Whitespace within the pattern is ignored, except when in a character class or preceded by an un-escaped backslash. When a line contains a `#`, neither in a character class nor preceded by an un-escaped backslash, all characters from the left-most such `#` through to the end of the line are ignored.

Table 6.3: A list of the flags accepted by the =∼ operator. Most are rarely used, but the `g` flag is very useful.

```
twister =~ s's'th'
```

Flags can be passed, as in `sed` or `perl`, to modify the precise behaviour of the regular expression. In the following example the `g` flag is used to perform a global search-and-replace of all instances of the letter `s` with the letters `th`:

```
pyxplot> twister="seven silver soda syphons"
pyxplot> twister =~ s/s/th/g
pyxplot> print twister
theven thilver thoda thyphonth
```

Table 6.3 lists all of the regular expression flags recognised by the =∼ operator.

6.3 Lists

List objects hold ordered sequences of other Pyxplot objects, which may include lists and dictionaries to create hierarchical data structures. They are created by enclosing a comma-separated list of objects by square brackets.

For example:

```
a = [10,colors.green,"bottles"]
```

Once created, more items can be added to a list using its `append(item)` and `insert(n,item)` methods, where the latter inserts an item at position n:

```
pyxplot> theFive = ["Cui","Mussorgsky"]
pyxplot> theFive.append("Borodin")
["Cui", "Mussorgsky", "Borodin"]
pyxplot> theFive.insert(0,"Balakirev")
["Balakirev", "Cui", "Mussorgsky", "Borodin"]
```

```
pyxplot> theFive.insert(-2,"Rimsky-Korsakov")
["Balakirev", "Cui", "Mussorgsky", "Rimsky-Korsakov", "Borodin"]
```

A complete list of the methods available on lists (itself a list of strings) can be found by calling the method `[].methods()`; they are also listed in Section 13.10. As with string methods, documentation of list methods is returned if the method object is printed:

```
pyxplot> print [].append
append(x) appends the object x to a list.
pyxplot> print [].insert
insert(n,x) inserts the object x into a list at position n.
```

Most methods that operate on lists, for example, append, extend and sort operations, return the list as their output. Unless this is stored in a variable, Pyxplot prints this return value to the terminal. In some cases this is useful: in the example above, it allowed us to see how the list was changing when we called its `append()` and `insert()` methods. Often, however, this terminal spam is unwanted. The `call` command allows methods to be called without printing their output, which is discarded:

```
pyxplot> a = ["William I"]
pyxplot> call a.extend(["William II","Henry I"])
pyxplot> call a.insert(0,"Edgar II")
pyxplot> call a.insert(0,"Edward the Confessor")
pyxplot> print a
["Edward the Confessor", "Edgar II", "William I", "William II", "Henry I"]
```

6.3.1 Using lists as stacks

The following example demonstrates the use of a list as a stack; note that the last item added to the stack is the first one to be popped:

```
pyxplot> myList = []
pyxplot> myList.append("opening wardrobe")
["opening wardrobe"]
pyxplot> myList.append("opening sock drawer")
["opening wardrobe", "opening sock drawer"]
pyxplot> myList.append("taking of sock")
["opening wardrobe", "opening sock drawer", "taking of sock"]
pyxplot> while (myList.len()>0) { print "Undo "+myList.pop() ; }
Undo taking of sock
Undo opening sock drawer
Undo opening wardrobe
```

6.3.2 Using lists as buffers

The following example demonstrates the use of a list as a buffer in which the first item added to the stack is the first one to be popped:

```
pyxplot> myList = []
pyxplot> for (i=1; i<12; i++)
pyxplot>  {
pyxplot>   if prime(i) { call myList.append(i) ; }
pyxplot>  }
pyxplot> while (myList) { print myList.pop(0) ; }
2
3
5
7
11
```

The function `prime(x)` returns true if `x` is a prime number, and false otherwise. In the final line, we make use of the fact that a list tests true if it contains any items, or false if it is empty.

6.3.3 Sorting lists

Methods are provided for sorting data in lists. The simplest of these is the `sort()` method, which sorts the members into order of ascending value.[4] The `reverse()` method can be used to invert the order of the list afterwards if descending order is wanted.

```
pyxplot> a = [8,4,7,3,6,2]
pyxplot> print a.sort()
[2, 3, 4, 6, 7, 8]
pyxplot> print a.reverse()
[8, 7, 6, 4, 3, 2]
```

Custom sorting

Often, however, a custom ordering is wanted. The `sortOn(f)` method takes a function of two arguments as its input. The function `f(a,b)` should return -1 if `a` is to be placed before `b` in the sorted list, 1 if `a` is to be placed after `b` in the sorted list, and zero if the two elements have equal ranking.
The `cmp(a,b)` function is often useful in making comparison functions for use with the `sortOn(f)` method: it returns either -1, 0 or 1 depending on Pyxplot's default way of comparing two objects. In the example below, we pass it the magnitude of `a` and `b` to sort a list in order of magnitude.

```
pyxplot> absCmp(a,b) = cmp(abs(a),abs(b))
pyxplot> a = range(-8,8)
pyxplot> print a
vector(-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7)
pyxplot> a = a.list()
pyxplot> print a.sortOn(absCmp)
```

[4] Non-numeric items are assigned arbitrary but consistent values for the purposes of sorting. Booleans are always lower-valued than numbers, but numbers are lower-valued than lists. Longer lists are always higher valued than shorter lists; larger dictionaries are always higher-valued than smaller dictionaries.

```
[0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5, -6, 6, -7, 7, -8]
pyxplot> print a.sort()
[-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]
```

In this example, the `range(start,end,step)` function is used to generate a raster of values between -8 and 8. It outputs a vector, which is converted into a list using the vector's `list()` method. More information about vectors is in Section 6.5.

The subroutine command, which is often used to implement more complicated sorting functions, will be covered in Section 7.8. For example, the function used above could have been written:

```
subroutine absCmp(a,b)
 {
  return cmp(abs(a),abs(b))
 }
```

Sorting lists of lists

The `sortOnElement(n)` method can be used to sort a list of lists on the nth sub-element of each sublist.

```
pyxplot> b = []
pyxplot> b.append([1797,1828,"Schubert"])
[[1797, 1828, "Schubert"]]
pyxplot> b.append([1770,1827,"Beethoven"])
[[1797, 1828, "Schubert"], [1770, 1827, "Beethoven"]]
pyxplot> b.append([1756,1791,"Mozart"])
[[1797, 1828, "Schubert"], [1770, 1827, "Beethoven"], [1756, 1791, "Mozart"]]
pyxplot> print b.sortOnElement(0) # Order of birth
[[1756, 1791, "Mozart"], [1770, 1827, "Beethoven"], [1797, 1828, "Schubert"]]
pyxplot> print b.sortOnElement(1) # Order of death
[[1756, 1791, "Mozart"], [1770, 1827, "Beethoven"], [1797, 1828, "Schubert"]]
pyxplot> print b.sortOnElement(2) # Alphabetical order
[[1770, 1827, "Beethoven"], [1756, 1791, "Mozart"], [1797, 1828, "Schubert"]]
```

6.3.4 Iterating over lists

The `foreach` command can be used to iterate over the members of a list; it will be covered in more detail in Section 7.3. The following example iterates over the words in a sentence:

```
pyxplot> poem = "An aged thrush, frail, gaunt, and small, "
pyxplot> poem+= "In blast-beruffled plume"
pyxplot> wordList = poem.split()
pyxplot> foreach word in wordList { print word ; }
An
aged
thrush,
```

```
frail,
gaunt,
and
small,
In
blast-beruffled
plume
```

6.3.5 Calling functions with lists of arguments

The `call(`f, a`)` function can be used to call a function with an arbitrary list of arguments. For example:

```
pyxplot> print romanNumeral(2012)
MMXII
pyxplot> print call(romanNumeral,[2012])
MMXII
pyxplot> print pow(2,8)
256
pyxplot> print call(pow,[2,8])
256
```

6.3.6 List mapping and filtering

The methods `filter(f)`, `map(f)` and `reduce(f)` can be used to perform actions on all of the members of a list in turn. `filter(f)` takes a function of one argument as its argument, and returns a new list of all of the members `x` of the original list for which `f(x)` tests true. For example:

```
pyxplot> txt = "once upon a time, there was a"
pyxplot> list = txt.split()
pyxplot> longWord(x) = x.len()>3
pyxplot> print list.filter(longWord)
["once", "upon", "time,", "there"]
```

The method `map(f)` also takes a function of one argument as its argument, and returns a list of the results `f(x)` for each of the members `x` of the original list. In other words, if `f` were `sin`, and the original list contained values of `x`, the result would be a list of values of `sin(x)`. This example converts a list of numbers into Roman numerals:

```
pyxplot> factors = primeFactors(1001)
pyxplot> print factors
[7, 11, 13]
pyxplot> romanFactors = factors.map(romanNumeral)
pyxplot> print romanFactors
["VII", "XI", "XIII"]
```

The method `reduce(f)` takes a function of two arguments as its argument. It first calls $f(a, b)$ on the first two elements of the list, and then continues

through the list calling $f(a, b)$ on the result and the next item in the list. The final result is returned:

```
pyxplot> multiply(x,y) = x*y
pyxplot> factors = primeFactors(1001)
pyxplot> print factors.reduce(multiply)
1001
```

6.3.7 Vectors versus lists

Vectors are similar to lists, except that all of their elements must be real numbers, and that all of the elements of any given vector must share common physical dimensions. Vectors are stored much more efficiently in memory than lists, since information about the types and physical units of each of the elements need not be stored. In addition they support a wide range of vector and matrix arithmetic operations.

Data from lists can also be plotted onto graphs, but the list must first be converted into a vector. See 6.5 for more information.

6.4 Dictionaries

Dictionaries, also known as associative arrays or content-addressable memories in other programming languages, store collections of objects, each of which has a unique name (or key). Objects are addressed by name, rather than by number:

```
pyxplot> myDict = {'red':colors.red, 'green':colors.green}
pyxplot> myDict['blue'] = colors.blue
pyxplot> print myDict['green']
cmyk(1,0,1,0)
pyxplot> call myDict.delete('green')
pyxplot> call myDict.delete('blue')
pyxplot> myDict['purple'] = colors.purple
pyxplot> print myDict
{"purple":cmyk(0.45,0.86,0,0), "red":cmyk(0,1,1,0)}
```

As the first line of this example shows, dictionaries can be created by enclosing a list of key–value pairs in curly brackets. As in python, a colon separates each key from its corresponding value, while the list of key–value pairs are comma-separated. That is, the general syntax is:

```
{ key1:value1 , key2:value2 , ... }
```

It is also possible to generate an empty dictionary, as {}. Items can later be referenced or assigned by name, where the name is placed in square brackets after the name of the dictionary. Items can be deleted with the dictionary's `delete(key)` method.

It is not an error to assign an item to a name which is already defined in the dictionary; the new assignment overwrites the old object with that name. It is, however, an error to attempt to access a key which is not defined in the

dictionary. The method `hasKey(key)` may be used to test whether a key is defined before attempting to access it.

Unlike in python, keys **must** be strings.

6.5 Vectors and matrices

Vectors are similar to lists, except that all of their elements must be real numbers, and that all of the elements of any given vector must share common physical dimensions. Vectors are stored much more efficiently in memory than lists, since information about the types and physical units of each of the elements need not be stored. In addition they support a wide range of vector and matrix arithmetic operations.

For example, applying the addition `+` operator to two lists concatenates the lists together, meanwhile the same operator applied to two vectors performs vector addition:

```
pyxplot> a = [1,2,3]
pyxplot> b = [4,0,6]
pyxplot> print a+b
[1, 2, 3, 4, 0, 6]
pyxplot> av = vector(a)
pyxplot> bv = vector(b)
pyxplot> print av+bv
vector(5, 2, 9)
```

In fact, whilst vectors do support the same `append` and `extend` methods as lists, to add either a single new element, or a list of new elements, to the end of the vector, these are very time consuming methods to run. It is much more efficient to create a vector of the desired length, and then to populate it with elements:

```
pyxplot> a = vector(10)*unit(m)
pyxplot> print a
vector(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)*unit(m)
pyxplot> a[5] = unit(inch)
pyxplot> print a
vector(0, 0, 0, 0, 0, 0.0254, 0, 0, 0, 0)*unit(m)
```

As the above example demonstrates, the `vector()` prototype can take not only a list or vector from which to make a vector object copy, but alternatively a single integer, which creates a zeroed vector of the specified length.

Similarly, the `matrix()` prototype can create a matrix from a list of lists, a list of vectors, a series of list arguments, a series of vector arguments, or two integers. In the final case, a zero matrix with the specified number of rows and columns is returned. If a matrix is specified as a series of vectors, these are taken to be the columns of the matrix; but if the matrix is specified as a series of lists, these are taken to be the rows of the matrix:

```
pyxplot> print matrix(3,4)
(0, 0, 0, 0)
```

```
(0, 0, 0, 0)
(0, 0, 0, 0)
pyxplot> print matrix([[1,2],[3,4]])
(1, 2)
(3, 4)
pyxplot> print matrix([vector(1,2),vector(3,4)])
(1, 3)
(2, 4)
```

Like vectors, matrices can have physical units, and adding two matrices together performs element-wise addition:

```
pyxplot> a = matrix([1,0],[0,1])*unit(s)
pyxplot> b = matrix([0,-1],[-1,0])*unit(s)
pyxplot> print a+b
(1, -1)
(-1, 1) *unit(s)
```

6.5.1 Dot and cross products

The dot product of two vectors can be found simply by multiplying the two vectors together:

```
pyxplot> a = vector(1,4)
pyxplot> b = vector(2,3)
pyxplot> print a*b
14
```

The cross product of two vectors, which is only defined for pairs of three-element vectors, can be found by passing the two vectors to the `cross(a,b)` function:

```
pyxplot> a = vector(1,4,1)
pyxplot> b = vector(2,3,2)
pyxplot> print cross(a,b)
vector(5, 0, -5)
```

6.5.2 Matrix algebra

Matrices can be multiplied by one another and by vectors to perform matrix arithmetic. This not only allows matrix equations to be solved, but also allows transformation matrices to be applied to vector positions on the vector graphics canvas. All of Pyxplot's vector graphics commands, which will be described in detail in Chapter 10, can accept positions as either comma-separated numerical components, or as vector objects. The following example demonstrates the use of a rotation matrix:

```
rotate(a) = matrix( [[cos(a),-sin(a)], \
                     [sin(a), cos(a)] ] )
pos = vector(0,5)*unit(cm)
theta = 30*unit(deg)
```

```
arrow from 0,0 to rotate(theta)*pos with linewidth 3
```

In addition to matrix multiplication, other arithmetic operations are available via the methods of matrix objects. Their methods `diagonal()` and `symmetric()` return `true` or `false` as appropriate. Their `size()` method returns the vector size of the matrix (rows, columns). Their `det()` method returns the determinate of the matrix and their `transpose()` method returns the matrix transpose.

Among more complex operations, `inv()` returns the inverse of a matrix, `eigenvalues()` returns a vector of the matrix's eigenvalues, and `eigenvectors()` returns a list of the matrix's corresponding eigenvectors.

6.5.3 Plotting data from vectors

Vectors can be used to pass calculated data to the `plot` command for plotting. Instead of supplying the name of a data file, or a function to be plotted, a series of colon-separated vector objects should be passed to the `plot` command. Each of the vectors should be the same length; the nth elements of each of the vectors are put together to form the columns of data for the nth data point.

The following example draws 100 random points on a graph:

```
N=100
a=vector(N) ; b=vector(N)
for i=0 to 99 { a[i]=random.random() ; }
for i=0 to 99 { b[i]=random.random() ; }
plot [0:1][0:1] a:b
```

Vectors support the same `filter()`, `map()` and `reduce()` methods as lists (see Section 6.3.6), and these can prove especially useful for preparing data for plotting. The following example selects fifty random points along the x-axis, and uses them to plot $\sin(x)$:

```
N=50
a=vector(N)
for i=0 to 99 { a[i]=random.random() ; }
b=a.map(sin)
plot [0:1][0:1] a:b
```

6.6 Colors

Most of Pyxplot's graph plotting and vector graphics commands have settings for specifying colors. A selection of widely-used colors may be specified by name, for example `red` and `blue`. However, greater freedom in choice of color is available by passing these commands objects of type `color`.

Several functions are available for making color objects:

- `gray(x)` returns a shade of gray. The argument x should be in the range 0–1. If $x = 0$, black is returned; if $x = 1$, white is returned.
- `rgb(r,g,b)` returns a color with the specified RGB components, which should be in the range 0–1.

- `cmyk(c,m,y,k)` returns a color with the specified CMYK components, which should be in the range 0–1.
- `hsb(h,s,b)` returns a color with the specified coordinates in hue–saturation–brightness color space, which should be in the range 0–1.

In addition, color objects corresponding to all of Pyxplot's built-in named colors can be found in the `colors` module.

```
a = colors.red
b = rgb(0,0.5,0)
box from 0,0 to 3,3 with color a fillcolor b lw 5
```

Once a color object has been made, various operations are supported. Multiplying or dividing a color by a number changes the brightness of the color. When two colors are compared, brighter colors are greater than darker colors. When two colors are added together, they are additively mixed in RGB space, so that adding red and green together produces yellow. When one color is subtracted from another, the opposite happens, so that yellow minus green is red.
The methods available on `color` objects are listed in Section 13.3.

6.6.1 Color representations of the electromagnetic spectrum

Two functions, in the `colors` module, provide color objects which approximate the color of particular wavelengths of light, or of electromagnetic spectra.

colors.wavelength(λ,$norm$)
The colors.wavelength(λ,$norm$) function returns a color representation of monochromatic light at wavelength λ, normalised to brightness $norm$. A value of $norm = 1$ is recommended for plotting the complete span of the electromagnetic spectrum without colors clipping to white.

colors.spectrum($spec, norm$)
The colors.spectrum($spec, norm$) function returns a color representation of the spectrum $spec$, normalised to brightness $norm$. $spec$ should be a function object that takes a single input (wavelength) with units of length, and may return an output with arbitrary units.
For an example of the use of these functions, see Section 8.12.

6.7 Dates

Pyxplot has a `date` object type which simplifies the process of working with dates and times. Pyxplot provides a range of pre-defined functions, in the `time` module, for creating and manipulating `date` objects. The functions for creating `date` objects are as follows:

time.fromCalendar($year, month, day, hour, min, sec$)
The time.fromCalendar($year, month, day, hour, min, sec$) function creates a date object from the specified calendar date. It takes six inputs: the year, the month

number (1–12), the day of the month (1–31), the hour of day (0–24), the number of minutes (0–59), and the number of seconds (0–59). To enter dates before AD 1, a year of 0 should be passed to indicate 1 BC, −1 should be passed to indicate the year 2 BC, and so forth. The `set calendar` command is used to change the current calendar.

time.fromJD(t)

The time.fromJD(t) function creates a date object from the specified numerical Julian date.

time.fromMJD(t)

The time.fromMJD(t) function creates a date object from the specified numerical modified Julian date.

time.fromUnix(t)

The time.fromUnix(t) function creates a date object from the specified numerical Unix time.

time.now()

The time.now() function creates a date object representing the present time.

The following example creates a date object representing midnight on 1st January 2000:

```
pyxplot> print time.fromCalendar(2000,1,1,0,0,0)
Sat 2000 Jan 1 00:00:00 UTC
pyxplot> a = time.fromCalendar(2000,1,1,0,0,0,"Australia/Perth")
pyxplot> print a # Note that this does not use Australian time
Fri 1999 Dec 31 15:59:59 UTC
pyxplot> set timezone "Pacific/Chatham"
pyxplot> print a
Sat 2000 Jan 1 05:45:00 +1345
pyxplot> set timezone "Antarctica/South_Pole"
pyxplot> print a
Sat 2000 Jan 1 05:00:00 NZDT
pyxplot> print a.toYear() # at the south pole
2000
pyxplot> print a.toYear("Europe/London")
1999
```

Once created, it is possible to add numbers with physical units of time to dates, as in the following example:

```
pyxplot> myDate = time.fromCalendar(2012,8,1,0,0,0)
pyxplot> print myDate + unit(7*day)
Wed 2012 Aug 8 00:00:00 UTC
pyxplot> print myDate - unit(2000*day)
Fri 2007 Feb 9 00:00:00 UTC
```

In addition, if one date is subtracted from another date, the time interval between the two dates is returned as a number with physical dimensions of time:

```
pyxplot> x = time.fromCalendar(-752,4,21,12,0,0)
pyxplot> y = time.fromCalendar( 476,9, 4,12,0,0)
pyxplot> print y-x
3.8764483e+10 s
pyxplot> print time.interval(y,x)
3.8764483e+10 s
pyxplot> print (y-x)/unit(year)
1228.3986
```

Standard string representations of calendar dates can be produced with the `print` command. It is also possible to use the string substitution operator, as in `"%s"%(date)`, or the `str` method of `date` objects, as in `date.str()`. In addition, the `time.string` function can be used to choose a custom display format for the date; for more information, see Section 4.11.

Several functions are provided for converting `date` objects back into various numerical forms of timekeeping and components of calendar dates:

toDayOfMonth()
The toDayOfMonth() method returns the day of the month of a date object in the current calendar.

toDayWeekName()
The toDayWeekName() method returns the name of the day of the week of a date object.

toDayWeekNum()
The toDayWeekNum() method returns the day of the week (1–7) of a date object.

toHour()
The toHour() method returns the integer hour component (0–23) of a date object.

toJD()
The toJD() method converts a date object to a numerical Julian date.

toMinute()
The toMinute() method returns the integer minute component (0–59) of a date object.

toMJD()
The toMJD() method converts a date object to a modified Julian date.

toMonthName()
The toMonthName() method returns the name of the month in which a date object falls.

toMonthNum()

The toMonthNum() method returns the number (1–12) of the month in which a date object falls.

toSecond()
The toSecond() method returns the seconds component (0–60) of a date object, including the non-integer component.

toUnix()
The toUnix() method converts a date object to a Unix time.

toYear()
The toYear() method returns the year in which a date object falls in the current calendar.

For example:

```
pyxplot> a = time.fromCalendar(2000,1,1,0,0,0)
pyxplot> time.string(a)
Sat 2000 Jan 1 00:00:00 UTC
pyxplot> time.string(a,"%d %B %Y")
1 January 2000
pyxplot> set calendar muslim
pyxplot> time.string(a,"%d %B %Y")
21 Dhu l-Qa'da 1389
```

More information on the manipulation of `date` objects can be found in Section 4.11.

6.8 Modules and classes

Modules provide a convenient way to group functions and variables together. Pyxplot's default functions are grouped into modules such as `os` and `random`. New modules can be created by calling the `module` object, which is a synonym for `types.module`. Once created, a module is like a dictionary, except that its elements can be accessed both as `module[item]` and more commonly as `module.item`. For example:

```
pyxplot> myFuncs = module()
pyxplot> myFuncs.squared(x) = x**2
pyxplot> myFuncs.cubed(x) = x**3
pyxplot> print myFuncs
module {
cubed          : cubed(x)=x**3
squared        : squared(x)=x**2
}
pyxplot> print myFuncs.squared(4)
16
pyxplot> print myFuncs['cubed'](2)
8
```

Modules can also serve as class prototypes. If a module is called like a

function, the return value is an *instance* of the module:

```
pyxplot> oldMan = module()
pyxplot> oldMan.info() = "Barefoot on the ice, \n\
.......> he staggers back and forth"
pyxplot> hurdyGurdyMan = oldMan()
pyxplot> hurdyGurdyMan.moreInfo() = "With numb fingers \n\
.......> he plays the best he can."
pyxplot> print hurdyGurdyMan.moreInfo()
With numb fingers \nhe plays the best he can.
pyxplot> print hurdyGurdyMan.info()
Barefoot on the ice, \nhe staggers back and forth
```

The module *instance* inherits all of the functions and variables of its parent object, but may also contain its own additional functions and variables, some of which may supersede those in the parent object if they have the same name. When functions or subroutines of a module instance are called, the special variable `self` is defined to equal the module instance object. This allows the function to store private data in the module instance, or to call other methods on the instance.

```
pyxplot> animal = module()
pyxplot> animal.info() = "I am a %s."%self.type
pyxplot> animal.moreInfo() = "My name is %s."%self.name
pyxplot> cat = animal()
pyxplot> cat.type = "cat"
pyxplot> subroutine cat.poke() { print "miaox!" ; }
pyxplot> cat.moreInfo() = "My name is %s."%self.name
pyxplot> tiddles = cat()
pyxplot> tiddles.name = "tiddles"
pyxplot> print tiddles.info()
I am a cat.
pyxplot> print tiddles.moreInfo()
My name is tiddles.
pyxplot> call tiddles.poke()
miaox!
```

As this example demonstrates, it is also possible to hierarchically instantiate modules: `tiddles` is an instance of `cat`, which is itself an instance of `animal`.

6.9 File handles

File handles provide a means of reading data directly from text files, or of writing data or logging information to files. Files are opened using the `open()` function:

open(x[,y])

The open(x[,y]) function opens the file x with string access mode y, and returns a file handle object.

The most commonly used access modes are `"r"`, to open a file read-only, `"w"`,

to open a file for writing, erasing any pre-existing file of the same filename, and `"a"`, to append data to the end of a file.

Alternatively, if what is wanted is a temporary scratch space, the `os.tmpfile()` function should be used:

os.tmpfile()
The os.tmpfile() function returns a file handle for a temporary file. The resulting file handle is open for both reading and writing.

The following methods are defined for file handles:

close()
The close() method closes a file handle.

dump(x)
The dump(x) method stores a typeable ASCII representation of the object x to a file. Note that this method has no checking for recursive hierarchical data structures.

eof()
The eof() method returns a boolean flag to indicate whether the end of a file has been reached.

flush()
The flush() method flushes any buffered data which has not yet physically been written to a file.

getPos()
The getPos() method returns a file handle's current position in a file.

isOpen()
The isOpen() method returns a boolean flag indicating whether a file is open.

read()
The read() method returns the contents of a file as a string.

readline()
The readline() method returns a single line of a file as a string.

readlines()
The readlines() method returns the lines of a file as a list of strings.

setPos(x)
The setPos(x) method sets a file handle's current position in a file.

write(x)
The write(x) method writes the string x to a file.

6.9.1 Storing data structures in text files

The `dump(x)` method of file handles is provided as a means of writing a typeable ASCII representation of the object `x` to file, for later recovery using the `load` command. It is similar to the `pickle()` function in Python.

There is no limit to the depth to which it will traverse hierarchically nested data structures, and will produce output of infinite length if there is recursive nesting.

Note that it is not able to store representations of function definitions or file handles, which are stored as null objects; class instances lose their relationship with their parents and are stored as free-standing modules.

Chapter 7

Programming: flow control

This chapter describes Pyxplot's facilities for automating repetitive tasks by using loops. At the end, we turn to Pyxplot's interaction with the shell and filing system in which it operates, introducing a simple framework for automatically re-executing Pyxplot scripts whenever they change, allowing plots to be automatically regenerated whenever the scripts used to produce them are modified.

7.1 Conditionals

The `if` statement can be used to conditionally execute a series of commands only when a certain criterion is satisfied. In its simplest form, its syntax is

```
if <expression> { .... }
```

where the expression can take the form of, for example, `x<0` or `y==1`. Note that the operator `==` is used to test the equality of two algebraic expressions; the operator `=` is only used to assign values to variables and functions. A full list of the operators available can be found in Table 3.1. As in many other programming languages, algebraic expressions are deemed to be true if they evaluate to any non-zero value, and false if they exactly equal zero. Thus, the following two examples are entirely legal syntax, and the first `print` statement will execute, but the second will not:

```
if 2*3 {
  print "2*3 is True"
 }

if 2-2 {
  print "2-2 is False"
 }
```

The variables `true` and `false` are predefined constants, making the following syntax legal:

```
if false {
  print "Never gets here"
 }
```

As in C, the block of commands which are to be conditionally executed is enclosed in braces (i.e. { }). The closing brace must be on a line by itself at the end of the block, or separated from the last command in the block by a semi-colon.

```
if (x==0)
 {
  print "x is zero"
 }

if (x==0) { print "x is zero" ; }
```

After such an `if` clause, it is possible to string together further conditions in `else if` clauses, perhaps with a final `else` clause, as in the example:

```
if (x==0)
 {
  print "x is zero"
 } else if (x>0) {
  print "x is positive"
 } else {
  print "x is negative"
 }
```

Here, as previously, the first script block is executed if the first conditional, `x==0`, is true. If this script block is not executed, the second conditional, `x>0`, is then tested. If this is true, then the second script block is executed. The final script block, following the `else`, is executed if none of the preceding conditionals have been true. Any number of `else if` statements can be chained one after another, and a final `else` statement can always be optionally supplied. The `else` and `else if` statements must always be placed on the same line as the closing brace of the preceding script block.

The precise way in which a string of `else if` statements are arranged in a Pyxplot script is a matter of taste: the following is a more compact but equivalent version of the example given above:

```
if      (x==0) { print "x is zero"     ; } \
else if (x> 0) { print "x is positive" ; } \
else           { print "x is negative" ; }
```

7.2 For loops

For loops may be used to execute a series of commands multiple times. Pyxplot allows `for` loops to follow either the syntax of the BASIC programming language, or the C syntax:

```
for <variable> = <start> to <end> [step <step>]
                            [loopname <loopname>]
  <code>

for (<initialise>; <criterion>; <step>)
  <code>
```

Here, `<code>` may be substituted by any block of Pyxplot commands enclosed in braces {}. The closing brace must be on a new line after the last command of the block.

The first form is similar to how the `for` command works in BASIC. The first time that the script block is executed, `variable` has the value `start`. Upon each iteration of the loop, it is incremented by amount `step`. The loop finishes when the value exceeds `end`. If `step` is negative, then `end` is required to be less than or equal to `start`. A step size of zero is considered to be an error. The iterator variable can have any physical dimensions, so long as `start`, `end` and `step` all have the same dimensions, but the iterator variable must always be a real number. If no step size is given then a step size of unity is assumed. As an example, the following script would print the numbers 0, 2, 4, 6 and 8:

```
for x = 0 to 10 step 2
 {
  print x
 }
```

In the C form of the `for` command, three expressions are provided, separated by semicolons. These are evaluated (a) when the loop initialises, (b) as a boolean test of whether the loop should continue iterating, and (c) at the end of each iteration, usually to increment/decrement variables as required. For example:

```
for (i=1,j=1; i<=256; i*=2,j++) { print "%3d %3d"%(j,i); }
```

The syntax

```
for (a; b; c) { ... ; }
```

is *almost* equivalent to

```
a; while (b) { ... ; c ; }
```

with the single exception that `continue` statements behave slightly differently. In the C form of the `for` command, the `continue` statement executes the expression `c` before the next iteration is started, even though the `while` loop above would not.

The optional `loopname` which can be specified in the `for` statement is used in conjunction with the `break` and `continue` statements which will be introduced in Section 7.6.

7.3 Foreach loops

Foreach loops may be used to run a script block once for each item in a list or dictionary. Alternatively, if a string is supplied, it is treated as a filename wildcard, and all matching files are returned. For example:

```
foreach x in [-1,pi,10]
 { print x ; }

foreach x in "*.dat"
```

```
 { print x ; }

myDict = { 'a':1 , 'b':2 }
foreach x in myDict
 { print x ; }
```

The first of these loops would iterate three times, with the variable `x` holding the values -1, π and 10 in turn. The second of these loops would search for any data files in the user's current directory with filenames ending in `.dat` and iterate for each of them. As previously, the wildcard character `*` matches any string of characters, and the character `?` matches any single character. Thus, `foo?.dat` would match `foo1.dat` and fooX.dat, but not `foo.dat` or `foo10.dat`. The effect of the `print` statement in this particular example would be rather similar to typing:

```
!ls *.dat
```

An error is returned if there are no files in the present directory which match the supplied wildcard. The following example would produce plots of all of the data files in the current directory with filenames `foo_*.dat` or `bar_*.dat` as `eps` files with matching filenames:

```
set terminal eps
foreach x in "foo_*.dat" "bar_*.dat"
 {
  outfilename =  x
  outfilename =~ s/dat/eps/
  set output outfilename
  plot x using 1:2
 }
```

If a dictionary is supplied to loop over, then the loop variable iterates over each of the keys in the dictionary.

7.4 Foreach datum loops

Foreach datum loops are similar to foreach loops in that they run a script block once for each item in a list. In this case, however, the list in question is the list of data points in a data file, samples of a function, or values in a vector. The syntax of the `foreach datum` command is similar to that of the commands met in the previous chapter for acting on data files: the standard modifiers `every`, `index`, `select` and `using` can be used to select which columns of the data file, and which subset of the datapoints, should be used:

```
foreach datum i,j,name in "data.dat" using 1:2:"%s"%($3)
  <code>

foreach datum x,y,z in sin(x):cos(x)
  <code>
```

```
foreach datum a,b in vector_a:vector_b
  <code>
```

The `foreach datum` command is followed by a comma-separated list of the variable(s) which are to be read from the input data on each iteration of the loop. The `using` modifier specifies the columns or rows of data which are to be used to set the values of each variable. In the first example above, the third variable, `name`, is set as a string, indicating that it will be set to equal whatever string of text is found in the third column of the data file.

Example 12: Calculating the mean and standard deviation of data.

The following Pyxplot script calculates the mean and standard deviation of a set of data points using the `foreach datum` command:

```
N_data = 0
sum_x = 0
sum_x2 = 0

foreach datum x in '--'
 {
  N_data ++
  sum_x += x
  sum_x2 += x**2
 }
1.3
1.2
1.5
1.1
1.3
END

mean = sum_x / N_data
SD = sqrt(sum_x2 / N_data - mean**2)

print "Mean = %s"%mean
print "SD = %s"%SD
```

For the data supplied, a mean of 1.28 and a standard deviation of 0.133 are returned.

7.5 While and do loops

The `while` command may be used to continue running a script block until some stopping criterion is met. Two types of while loop are supported:

```
while <criterion> [ loopname <name> ]
 {
```

```
  ....
 }

do [ loopname <name> ]
 {
  ....
 } while <criterion>
```

In the former case, the enclosed script block is executed repeatedly, and the algebraic expression supplied to the `while` command is tested immediately before each repetition. If it tests false, then the loop finishes. The latter case is very similar, except that the supplied algebraic expression is tested immediately *after* each repetition. Thus, the former example may never actually execute the supplied script block if the looping criterion tests false on the first iteration, but the latter example is always guaranteed to run its script block at least once.

The following example would continue looping indefinitely until stopped by the user, since the value `1` is considered to be true:

```
while (1)
 {
  print "Hello, world!"
 }
```

7.6 The `break` and `continue` statements

The `break` and `continue` statements may be placed within loop structures to interrupt their iteration. The `break` statement terminates execution of the smallest loop currently being executed, and Pyxplot resumes execution at the next statement after the closing brace which marks the end of that loop structure. The `continue` statement terminates execution of the *current iteration* of the smallest loop currently being executed, and execution proceeds with the next iteration of that loop, as demonstrated by the following pair of examples:

```
pyxplot> for i=0 to 4
pyxplot>  {
pyxplot>   if (i==2) { break ; }
pyxplot>   print i
pyxplot>  }
0
1
pyxplot> for i=0 to 4
pyxplot>  {
pyxplot>   if (i==2) { continue ; }
pyxplot>   print i
pyxplot>  }
0
1
3
4
```

Note that if several loops are nested, the `break` and `continue` statements only act on the innermost loop. If either statement is encountered outside of a loop structure, an error results. Optionally, the `for`, `foreach`, `do` and `while` commands may be supplied with a name for the loop, prefixed by the word `loopname`, as in the examples:

```
for i=0 to 4 loopname iloop
...
foreach i in "*.dat" loopname DatafileLoop
...
```

When loops are given such names, the `break` and `continue` statements may be followed by the name of the loop to be broken out of, allowing the user to act on loops other than the innermost one.

7.7 The conditional operator

The conditional operator provides a compact means of inserting conditional expressions. Following the syntax of C, it takes three arguments and is written as `a ? b : c`. The first argument, `a` is a truth criterion to be tested. If the criterion is true, then the operator returns its second argument `b` as its output. Otherwise, the function's third argument `c` is returned.

```
pyxplot> f(x) = (x>0)?x:0
pyxplot> print "%s %s %s %s %s"%(f(-2),f(-1),f(0),f(1),f(2))
0 0 0 1 2
pyxplot> x = 2
pyxplot> print "x is %s"%(x>0)?"positive":"negative"
positive
```

7.8 Subroutines

Subroutines are similar to mathematical functions (see Section 4.3), and once defined, can be used anywhere in algebraic expressions, just as functions can be. However, instead of being defined by a single algebraic expression, whenever a subroutine is evaluated, a block of Pyxplot commands of arbitrary length is executed. This gives much greater flexibility for implementing complex algorithms. Subroutines are defined using the following syntax:

```
subroutine <name>(<variable1>,...)
 {
  ...
  return <value>
 }
```

Where `name` is the name of the subroutine, `variable1` is an argument taken by the subroutine, and the value passed to the `return` statement is the value returned to the caller. Once the `return` statement is reached, execution of the subroutine is terminated. The following two examples would produce entirely equivalent results:

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

subroutine f(x,y)
 {
  return x*sin(y)
 }
```

In either case, the function/subroutine could be evaluated by typing:

```
print f(1,pi/2)
```

If a subroutine ends without any value being returned using the `return` statement, then a value of zero is returned.

Subroutines may serve one of two purposes. In many cases they are used to implement complicated mathematical functions for which no simple algebraic expression may be given. Secondly, they may be used to repetitively execute a set of commands whenever they are required. In the latter case, the subroutine may not have a return value, but may merely be used as a mechanism for encapsulating a block of commands. In this case, the `call` command may be used to execute a subroutine, discarding any return value which it may produce, as in the example:

```
pyxplot> subroutine f(x,y) { print "%s - %s = %s"%(x,y,x-y) ; }
pyxplot> call f(2,1)
2 - 1 = 1
pyxplot> call f(5*unit(inch), 10*unit(mm))
127 mm - 10 mm = 117 mm
```

Example 13: An image of a Newton fractal.

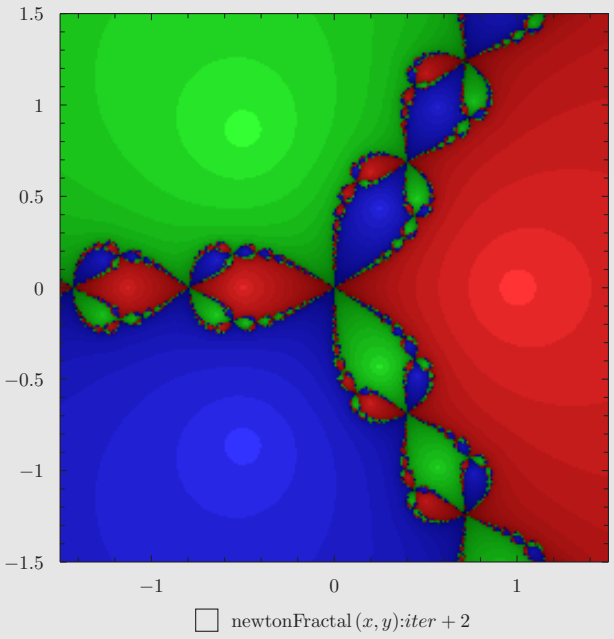
Newton fractals are formed by iterating the equation

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)},$$

subject to the starting condition that $z_0 = c$, where c is any complex number c and $f(z)$ is any mathematical function. This series is the Newton-Raphson method for numerically finding solutions to the equation $f(z) = 0$, and with time usually converges towards one such solution for well-behaved functions. The complex number c represents the initial guess at the position of the solution being sought. The Newton fractal is formed by asking which solution the iteration converges upon, as a function of the position of the initial guess c in the complex plane. In the case of the cubic polynomial $f(z) = z^3 - 1$, which has three solutions, a map might be generated with points colored red, green or blue to represent convergence towards the three roots.

If c is close to one of the roots, then convergence towards that particular root is guaranteed, but further afield the map develops a fractal structure. In this example, we define a Pyxplot subroutine to produce such a map as a function of $c = x + iy$, and then plot the resulting map using the `colormap` plot style (see Section 8.12). To make the fractal prettier – it contains, after all, only three colors as strictly defined – we vary the brightness of each point depending on how many iterations are required before the series ventures within a distance of $|z_n - r_i| < 10^{-2}$ of any of the roots r_i.

```
set numerics complex
set unit angle nodimensionless

root1 = exp(i*unit(  0*deg))
root2 = exp(i*unit(120*deg))
root3 = exp(i*unit(240*deg))

tolerance = 1e-2

subroutine newtonFractal(x,y)
{
global iter
z = x+i*y
iter = 0
while (1)
{
z = z - (z**3-1)/(3*z**2)
if abs(z-root1)<tolerance { ; return 1 ; }
if abs(z-root2)<tolerance { ; return 2 ; }
if abs(z-root3)<tolerance { ; return 3 ; }
iter = iter + 1
}
}

# Plot Newton fractal
set size square
set key below
set xrange [-1.5:1.5]
set yrange [-1.5:1.5]
set sample grid 250x250
set colmap hsb(c1*0.667,0.8+0.2*c2,1.0-0.8*c2)
set nocolkey
set log c2
plot newtonFractal(x,y):iter+2 with colormap
```



Example 14: The dynamics of the simple pendulum.

The equation of motion for a pendulum bob may be derived from the rotational analogue to Newton's Second Law, $G = I\ddot{\theta}$ where G is torque, I is moment of inertia and θ is the displacement of the pendulum bob from the vertical. For a pendulum of length l, with a bob of mass m, this equation becomes $-mgl\sin\theta = ml^2\ddot{\theta}$. In the small-angle approximation, such that $\sin\theta \approx \theta$, it reduces to the equation for simple harmonic motion, with the solution

$$\theta_{\text{approx}} = \omega \sin\left(\sqrt{\frac{g}{l}}t\right). \tag{7.1}$$

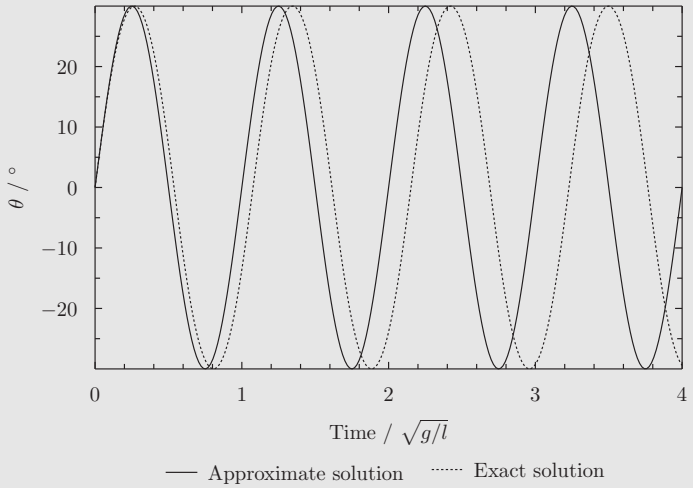
A more exact solution requires integration of the second-order differential equation of motion including the $\sin\theta$ term. This integral cannot be done analytically, but the solution can be written in the form

$$\theta_{\text{exact}}(t) = 2\sin^{-1}\left[k\,\text{sn}\left(\sqrt{\frac{g}{l}}t, k\right)\right]. \tag{7.2}$$

where $\text{sn}(u, m)$ is a Jacobi elliptic function and $k = \sin(\omega/2)$. The Jacobi elliptic function cannot be analytically computed, but can be numerically approximated using the `jacobi_sn(u,m)` function in Pyxplot.

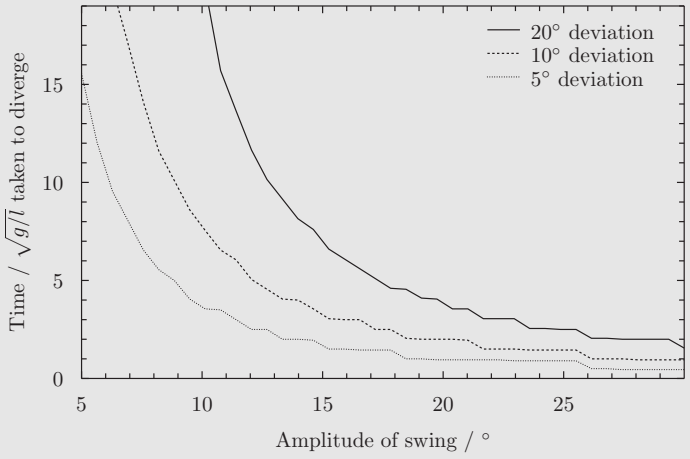
Below, we produce a plot of Equations (7.1) and (7.2). The horizontal axis is demarcated in units of the dimensionless period of the pendulum to eliminate g and l, and a swing amplitude of $\pm 30°$ is assumed:

```
set unit angle nodimensionless

theta_approx(a,t) = a*sin(2*pi*t)
theta_exact (a,t) = 2*asin(sin(a/2)*jacobi_sn(2*pi*t,sin(a/2)))

set unit of angle degrees
set key below
set xlabel r'Time / $\sqrt{g/l}$'
set ylabel r'$\theta$'
omega = unit(30*deg)
plot [0:4] theta_approx(omega,x) title 'Approximate solution', \
theta_exact (omega,x) title 'Exact solution'
```



As is apparent, at this amplitude, the exact solution begins to deviate noticeably from the small-angle solution within 2–3 swings of the pendulum. We now seek to quantify more precisely how long the two solutions take to diverge by defining a subroutine to compute how long T it takes before the two solutions to deviate by some amount ψ. We then plot these times as a function of amplitude ω for three deviation thresholds. Because this subroutine takes a significant amount of time to run, we only compute 40 samples for each value of ψ:

```
subroutine pendulumDivergenceTime(omega, deviation)
{
for t=0 to 20 step 0.05
{
approx = theta_approx(omega,t)
exact  = theta_exact (omega,t)
if (abs(approx-exact)>deviation) { ;break; }
}
return t
}

set key top right
set xlabel r'Amplitude of swing'
set ylabel r'Time / $\sqrt{g/l}$ taken to diverge'
set samples 40
plot [unit(5*deg):unit(30*deg)][0:19] \
pendulumDivergenceTime(x,unit(20*deg)) title r"$20^\circ$ deviation", \
pendulumDivergenceTime(x,unit(10*deg)) title r"$10^\circ$ deviation", \
pendulumDivergenceTime(x,unit( 5*deg)) title r"$ 5^\circ$ deviation"
```



7.9 Macros

The @ operator can be used for literal substitution of the content of a string variable into the command line. The name of the string variable follows the @ sign, and its content is expanded to the command line, as in this example

```
mac = "with lines lw 2 lt 1"
plot sin(x) @mac
```

which is equivalent to

```
plot sin(x) with lines lw 2 lt 1
```

The macro, being a string, can contain any characters, but as with other variable names, the name of the macro can contain only alphanumeric characters

and the underscore sign. This also means that any operator, with the exception of the `and` and `or` operators, can signify the end of the macro name, without the need for a trailing white space. Therefore, in the example

```
foo = "50"
print @foo*3
```

the end result is 150; `50*3` is passed to the command line interpreter.

7.10 The exec command

The `exec` command can be used to execute Pyxplot commands contained within string variables. For example:

```
terminal="eps"
exec "set terminal %s"%(terminal)
```

It can also be used to write obfuscated Pyxplot scripts, and its use should be minimized wherever possible.

7.11 Assertions

The `assert` command can be used to assert that a logical expression, such as `x>0`, is true. An error is reported if the expression is false, and optionally a string can be supplied to provide a more informative error message to the user:

```
assert x>0
assert y<0 "y must be less than zero."
```

The `assert` command can also be used to test the version number of Pyxplot. It is possible to test either that the version is newer than or equal to a specific version, using the `>=` operator, or that it is older than a specific version, using the `<` operator, as demonstrated in the following examples:

```
assert version >= 0.8.2
assert version <  0.8  "This script is designed for Pyxplot 0.7"
```

7.12 Raising exceptions

Pyxplot's `raise(e,s)` function is used to raise exceptions when error conditions are met. Its first argument `e` specifies the type of exception, and should be an object of type `exception`. The second argument should be an error message string. Pyxplot has a range of default exception types, which can be found as `exception` objects in the module `exceptions`. Alternatively, the object `types.exception` may be called with a single string argument to make a new exception type. For example:

```
raise(exceptions.syntax , "Input could not be parsed")

a=types.exception("user error")
raise(a, "The user made a mistake")
```

Alternatively, `exception` objects have a method `raise(s)` which can be called as follows:

```
a=types.exception("user error")
a.raise("The user made a mistake")
```

7.13 Shell commands

Shell commands may be executed directly from within Pyxplot by prefixing them with an `!` character. The remainder of the line is sent directly to the shell, for example:

```
!ls -l
```

Semi-colons cannot be used to place further Pyxplot commands after a shell command on the same line.

✗
```
!ls -l ; set key top left
```

It is also possible to substitute the output of a shell command into a Pyxplot command. To do this, the shell command should be enclosed in back-quotes (‘), as in the following example:

```
a=‘ls -l *.ppl | wc -l‘
print "The current directory contains %d Pyxplot scripts."%(a)
```

It should be noted that back-quotes can only be used outside quotes. For example,

✗
```
set xlabel ’‘ls‘’
```

will not work. One way to do this would be

```
set xlabel ‘echo "’" ; ls ; echo "’"‘
```

a better way would be to use the `os.system` or `os.popen` functions:

✔
```
fileList = os.popen("ls","r").read()
set xlabel fileList
```

Note that it is not possible to change the current working directory by sending the `cd` command to a shell, as this command would only change the working directory of the shell in which the single command is executed:

✗
```
!cd ..
```

Pyxplot has its own `cd` command for this purpose, as well as its own `pwd` command:

✔
```
cd ..
```

7.14 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.ppl
```

would set `pyxplot_watch` to watch the command script file `script.ppl`. 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.[1]

This is especially useful when combined with ghostview's watch facility. For example, suppose that a script `foo.ppl` 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.ppl
```

[1] 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.

Part II

Plotting and vector graphics

Chapter 8

Plotting: a complete guide

This part of the manual provides a complete description of Pyxplot's commands for producing graphs and vector graphics. This chapter extends the overview of the `plot` command in Chapter 3, providing a systematic description of how the appearance of plots can be configured. Subsequent chapters describe how to produce graphical output in a range of image formats (Chapter 9), how to produce galleries of multiple plots side-by-side, and how to produce more sophisticated vector graphics (Chapter 10).

8.1 The `with` modifier

In Chapter 3 an overview of the syntax of the `plot` command was provided, including the `every`, `index`, `select` and `using` modifiers, which can be used to control *which* data should be plotted. The `with` modifier controls *how* data should be plotted. For example, the statement

```
plot "data.dat" index 1 using 4:5 with lines
```

specifies that data should be plotted with lines connecting each data point to its neighbors. We term the keyword `lines` a *plot style*. The `with` modifier can also be followed by a variety of settings which fine-tune aspects of how data are displayed. For example, the statement

```
plot "data.dat" with lines linewidth 2.0
```

would connect data points with a line of twice the default width.

The next section will provide a complete list of all of Pyxplot's plot styles – i.e. the words which may be used in place of `lines`. First we list all of the modifiers such as `linewidth` which may be used to alter the exact appearance of these plot styles. These are as follows:

- `color` – used to select the color in which the dataset is to be plotted. It should be followed either by a number, to select a color from the present palette (see Section 8.1.1); by a recognised color name, a complete list of which can be found in Section 19.4; or by a color object, such as may be created by the functions `gray(g)`, `rgb(r,g,b)`, `cmyk(c,m,y,k)` or `hsb(h,s,b)`. This modifier may also be spelt `colour`.

- `fillcolor` – used to select the color in which the dataset is filled. The color may be specified using any of the styles listed for `color`. May also be spelt `fillcolour`.

- `linetype` – used to numerically select the type of line – for example, solid, dotted, dashed, etc. – which should be used by line-based plot styles. A complete list of Pyxplot's numbered line types can be found in Chapter 18. May be abbreviated `lt`.

- `linewidth` – used to select the width of line which should be used by line-based plot styles, where unity represents the default width. May be abbreviated `lw`.

- `pointlinewidth` – used to select the width of line which should be used to stroke points in point-based plot styles, where unity represents the default width. May be abbreviated `plw`.

- `pointsize` – used to select the size of drawn points, where unity represents the default size. May be abbreviated `ps`.

- `pointtype` – used to numerically select the type of point – for example, crosses, circles, etc. – used by point-based plot styles. A complete list of Pyxplot's numbered point types can be found in Chapter 18. May be abbreviated `pt`.

Any number of these modifiers may be placed sequentially after the keyword `with`, as in the following examples:

```
plot 'datafile' using 1:2 with points pointsize 2
plot 'datafile' using 1:2 with lines color red linewidth 2
plot 'datafile' using 1:2 with lp col 1 lw 2 ps 3
```

Where modifiers take numerical values, expressions of the form `$2+1`, similar to those supplied to the `using` modifier, may be used to read numbers from the supplied data set. In this case, each datapoint will be displayed in a different style or in a different color (in the example given, depending on the values in the second column of the supplied data).

The following example would plot a data file with `points`, drawing the position of each point from the first two columns of the supplied data file and the size of each point from the third column:

```
plot 'datafile' using 1:2 with points pointsize $3
```

Not all of these modifiers are applicable to all of Pyxplot's plot styles. For example, the `linewidth` modifier has no effect on plot styles which do not draw lines between datapoints. Where modifiers are applied to plot styles for which they have no defined effect, the modifier has no effect, but no error results. Table 8.1 lists which modifiers act on which plot styles.

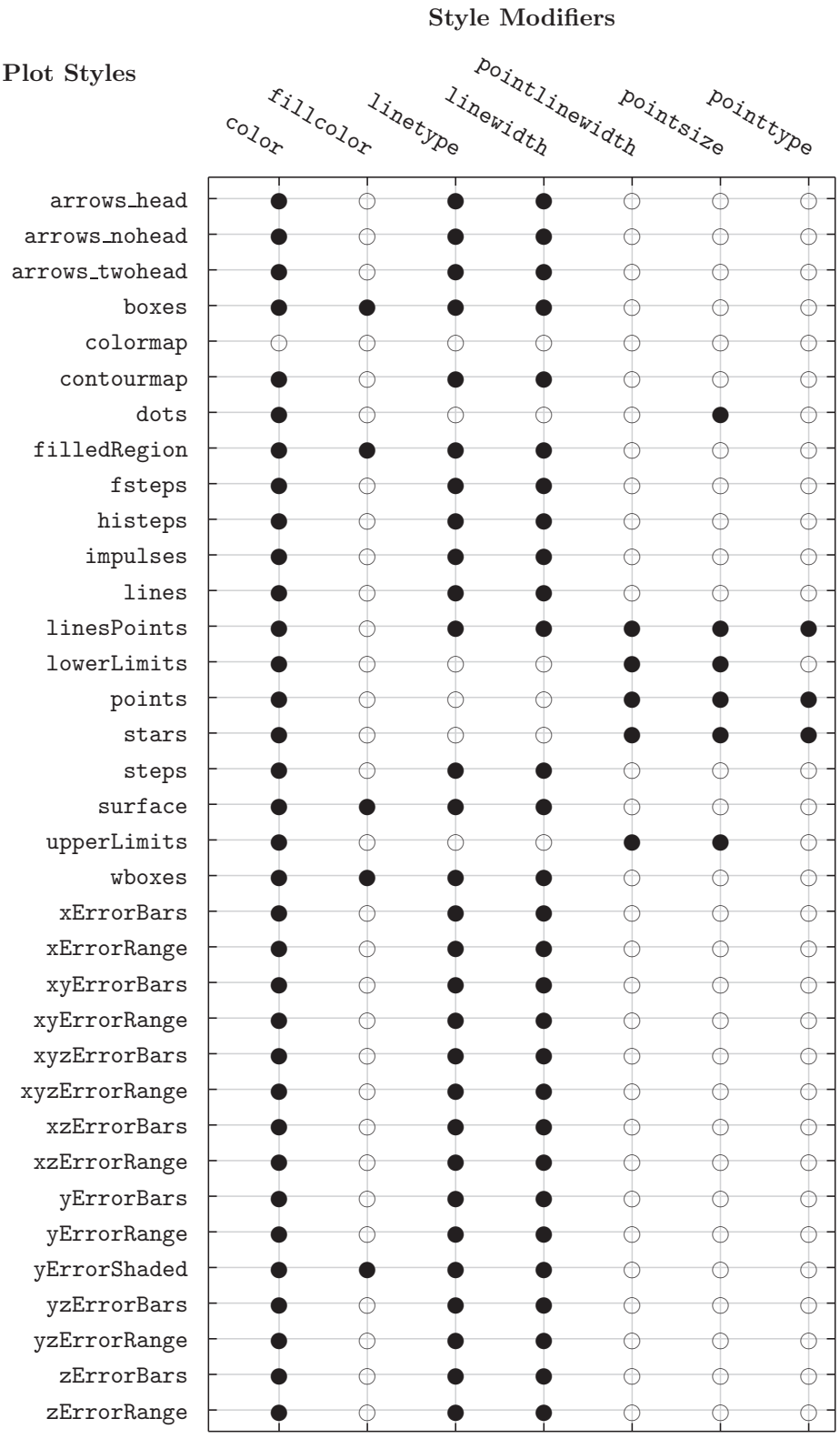


Table 8.1: A list of the plot styles affected by each style modifiers.

8.1.1 The palette

Wherever Pyxplot takes a color as an input to a command, the user has three options for how it may be specified. A selection of widely-used colors may be specified by name, for example `red` and `blue`. A complete list of such colors may be found in Section 19.4. Alternatively, an object of type `color` may be provided, such as `rgb(0,1,0)`, `hsb(0.5,0.5,0.5)`, `gray(0.2)`, `colors.green + colors.red`, or `colors.yellow - colors.green`.

The third option is to specify a numbered color from Pyxplot's *palette*. By default, this contains a series of visually distinctive colors which are, insofar as possible, also distinctive to users with most common forms of color blindness:

1 black	2 red	3 blue	4 magenta	5 cyan	6 brown
7 salmon	8 gray	9 green	10 navyBlue	11 periwinkle	12 pineGreen
13 seaGreen	14 greenYellow	15 orange	16 carnationPink	17 plum	

The first color is number 1, the second number 2, and so forth. As well as being accessible by number, these colors also form the default series which Pyxplot chooses for successive datasets when their colors are not individually specified.

The current palette may be queried using the `show palette` command, and changed using the `set palette` command, which takes a comma-separated list of colors, as in the example:

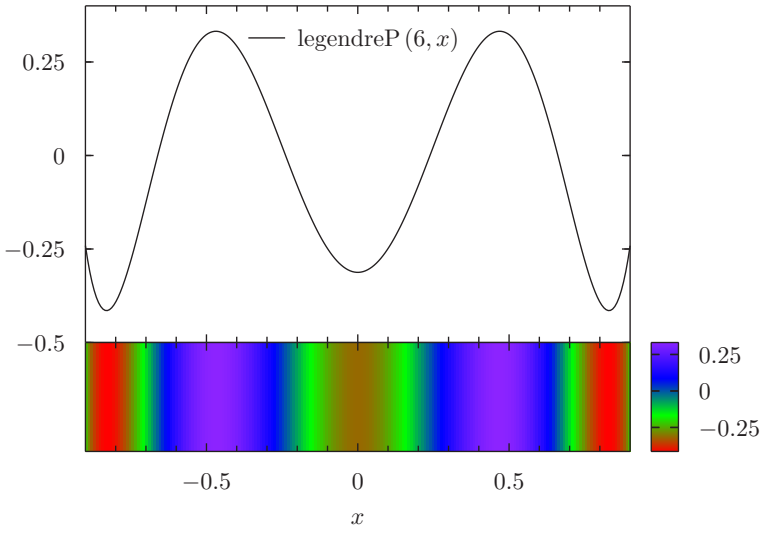
```
set palette brickRed, limeGreen, cadetBlue
```

The palette is treated as a cyclic list, and so in the above example, color number 4 would map to `brickRed`, as would color numbers 1 and 8. The default palette which Pyxplot uses on startup may be changed by setting up a configuration file, as described in Chapter 19.

If a non-integer color is requested from the palette, for example color 1.5, then a color is returned which is half-way in between colors 1 and 2 in RGB space; in this case, brown. This can be used to produce custom color gradients, as the following example demonstrates (the `colormap` plot style will be described in Section 8.12):

```
set multiplot
set width 8
set xformat ''
set key xcenter
plot [-0.9:0.9] legendreP(6,x)
set size 8 ratio 0.2
set origin 0,-1.6
set nokey
set noytics
unset xformat
set xlabel '$x$'
set palette red , green , blue , purple
set colmap 3*c1+1
```

```
set c1tics -1,0.25
set sample grid 200x2
plot [-0.9:0.9] legendreP(6,x) with colormap
```



8.1.2 Default settings

In addition to setting these parameters on a per-dataset basis, the `linewidth`, `pointlinewidth` and `pointsize` settings can also have their default values changed for all datasets as in the following examples:

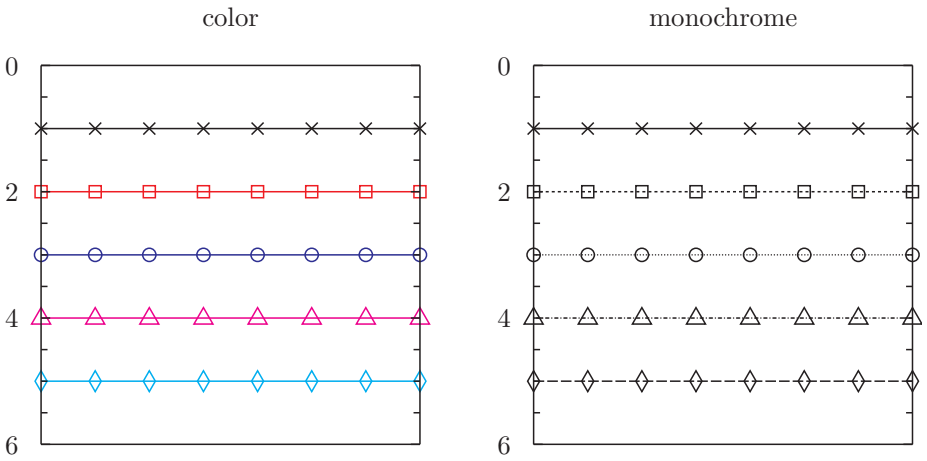
```
set linewidth 1
set pointlinewidth 2
set pointsize 3
plot "datafile"
```

In each case, the normal default values of these settings are 1. The default values of the `color`, `linetype` and `pointtype` settings depend whether the current graphic output device is set to produce color or monochrome output (see Chapter 9.1).

In the case of color output, the colors of each of the comma-separated datasets plotted on a graph are drawn sequentially from the currently-selected palette, and all lines are drawn as solid lines (`linetype 1`). The symbols used to draw each dataset are drawn sequentially from Pyxplot's available point types. In the case of monochrome output, all datasets are plotted in black and both the line types and point types used to draw each dataset are drawn sequentially from Pyxplot's available options.

The following simple example demonstrates this:

```
set terminal color
plot [][6:0] 1 with lp, 2 with lp, 3 w lp, 4 w lp, 5 w lp
set terminal monochrome
replot
```



8.2 Pyxplot's plot styles

This section provides a complete list of Pyxplot's *plot styles*, arranged into groups for clarity. Table 8.2 summarises the columns of data expected by each plot style when used on two- and three-dimensional plots. The following sections describe each of these plot styles in turn.

8.2.1 Lines and points

The following is a list of Pyxplot's simplest plot styles, all of which take two columns of input data on 2D plots (three columns on 3D plots), which represent the x-, y- (and z-)coordinates of the positions of each point:

- `dots` – places a small dot at each datum.
- `lines` – connects adjacent data points with straight lines.
- `linespoints` – a combination of both lines and points.
- `lowerlimits` – places a lower-limit sign (⊥) at each datum.
- `points` – places a marker symbol at each datum.
- `stars` – similar to `points`, but uses a different set of marker symbols, based on the stars drawn in Johann Bayer's highly ornate star atlas *Uranometria* of 1603.
- `upperlimits` – places an upper-limit sign (⊤) at each datum.

Example 15: A Hertzsprung-Russell diagram.

Style	Columns (2D plots)	Columns (3D plots)
arrows_head	(x_1, y_1, x_2, y_2)	$(x_1, y_1, z_1, x_2, y_2, z_2)$
arrows_nohead	(x_1, y_1, x_2, y_2)	$(x_1, y_1, z_1, x_2, y_2, z_2)$
arrows_twohead	(x_1, y_1, x_2, y_2)	$(x_1, y_1, z_1, x_2, y_2, z_2)$
boxes	(x, y)	(x, y)
colormap	$(x, y, c_1, \ldots)$	$(x, y, c_1, \ldots)$
contourmap	$(x, y, c_1, \ldots)$	$(x, y, c_1, \ldots)$
dots	(x, y)	(x, y, z)
FilledRegion	(x, y)	(x, y)
fsteps	(x, y)	(x, y)
histeps	(x, y)	(x, y)
impulses	(x, y)	(x, y, z)
lines	(x, y)	(x, y, z)
LinesPoints	(x, y)	(x, y, z)
LowerLimits	(x, y)	(x, y, z)
points	(x, y)	(x, y, z)
stars	(x, y)	(x, y, z)
steps	(x, y)	(x, y)
surface	(x, y, z)	(x, y, z)
UpperLimits	(x, y)	(x, y, z)
wboxes	(x, y, w)	(x, y, w)
XErrorBars	(x, y, σ_x)	(x, y, z, σ_x)
XErrorRange	$(x, y, x_{\min}, x_{\max})$	$(x, y, z, x_{\min}, x_{\max})$
XYErrorBars	$(x, y, \sigma_x, \sigma_y)$	$(x, y, z, \sigma_x, \sigma_y)$
XYErrorRange	$(x, y, x_{\min}, x_{\max}, y_{\min}, y_{\max})$	$(x, y, z, x_{\min}, x_{\max}, y_{\min}, y_{\max})$
XYZErrorBars	$(x, y, z, \sigma_x, \sigma_y, \sigma_z)$	$(x, y, z, \sigma_x, \sigma_y, \sigma_z)$
XYZErrorRange	$(x, y, z, x_{\min}, x_{\max}, y_{\min}, -$ $- y_{\max}, z_{\min}, z_{\max})$	$(x, y, z, x_{\min}, x_{\max}, y_{\min}, -$ $- y_{\max}, z_{\min}, z_{\max})$
XZErrorBars	$(x, y, z, \sigma_x, \sigma_z)$	$(x, y, z, \sigma_x, \sigma_z)$
XZErrorRange	$(x, y, z, x_{\min}, x_{\max}, z_{\min}, z_{\max})$	$(x, y, z, x_{\min}, x_{\max}, z_{\min}, z_{\max})$
YErrorBars	(x, y, σ_y)	(x, y, z, σ_y)
YErrorRange	$(x, y, y_{\min}, y_{\max})$	$(x, y, z, y_{\min}, y_{\max})$
YErrorShaded	$(x, y_{\min}, y_{\max})$	$(x, y_{\min}, y_{\max})$
YZErrorBars	$(x, y, z, \sigma_y, \sigma_z)$	$(x, y, z, \sigma_y, \sigma_z)$
YZErrorRange	$(x, y, z, y_{\min}, y_{\max}, z_{\min}, z_{\max})$	$(x, y, z, y_{\min}, y_{\max}, z_{\min}, z_{\max})$
ZErrorBars	(x, y, z, σ_z)	(x, y, z, σ_z)
ZErrorRange	$(x, y, z, z_{\min}, z_{\max})$	$(x, y, z, z_{\min}, z_{\max})$

Table 8.2: A summary of the columns of data expected by each of Pyxplot's plot styles when used on two- and three-dimensional plots.

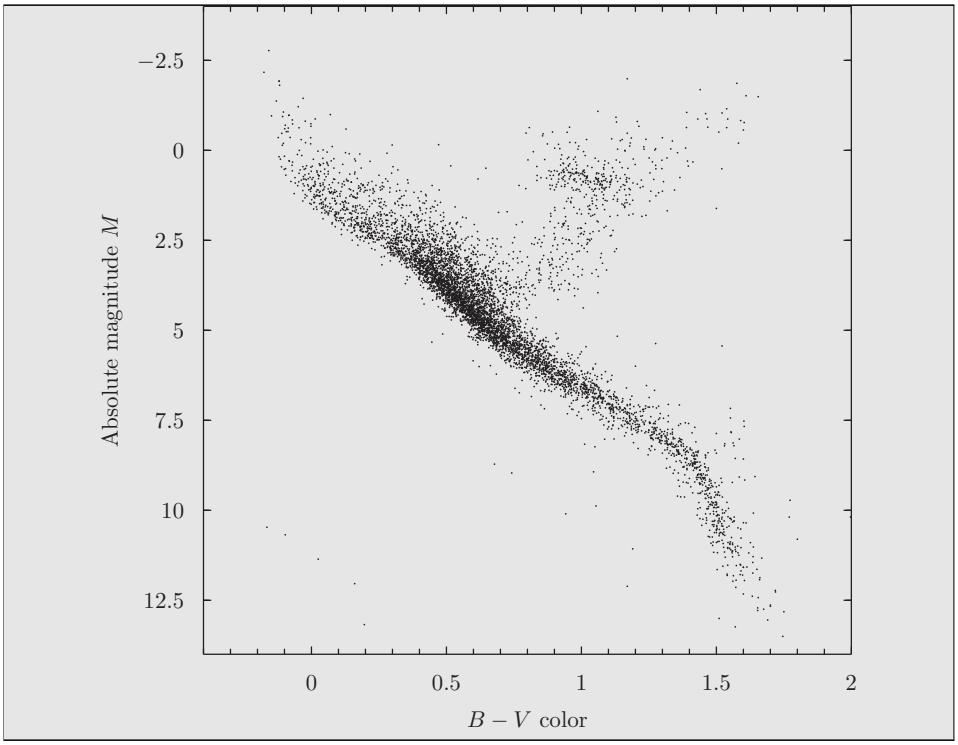
Hertzsprung-Russell (HR) diagrams are scatter-plots of the luminosities of stars plotted against their colors, on which most normal stars lie along a tight line called the main sequence, whilst unusual classes of stars – giants and dwarfs – can be readily identified on account of their not lying along this main sequence. The principal difficulty in constructing accurate HR diagrams is that the luminosities L of stars can only be calculated from their observed brightnesses F, using the relation $L = Fd^2$ if their distances d are known. In this example, we construct an HR diagram using observations made by the European Space Agency's *Hipparcos* spacecraft, which accurately measured the distances of over a million stars between 1989 and 1993.

The Hipparcos catalogue can be downloaded for free from `ftp://cdsarc.u-strasbg.fr/pub/cats/I/239/hip_main.dat.gz`; a description of the catalogue can be found at `http://cdsarc.u-strasbg.fr/viz-bin/Cat?I/239`. In summary, though the data is arranged in a non-standard format which Pyxplot cannot read without a special input filter, the following Python script generates a text file with four columns containing the magnitudes m, $B-V$ colors and parallaxes p of the stars, together with the uncertainties in the parallaxes. From these values, the absolute magnitudes M of the stars – a measure of their luminosities – can be calculated using the expression $M = m + 5\log_{10}\left(10^2 p\right)$, where p is measured in milli-arcseconds:

```
for line in open("hip_main.dat"):
 try:
  Vmag = float(line[41:46])
  BVcol = float(line[245:251])
  parr = float(line[79:86])
  parre = float(line[119:125])
  print "%s,%s,%s,%s"%(Vmag, BVcol, parr, parre)
 except ValueError:  pass
```

The resultant four columns of data can then be plotted in the `dots` style using the following Pyxplot script. Because the catalogue is very large, and many of the parallax datapoints have large errorbars producing large uncertainties in their vertical positions on the plot, we use the `select` statement to pick out those datapoints with parallax signal-to-noise ratios of better than 20.

```
set nokey
set size square
set xlabel '$B-V$ color'
set ylabel 'Absolute magnitude $M$'
plot [-0.4:2][14:-4] 'hrdiagram.dat.gz' w d ps 3
```



8.2.2 Error bars

The following pair of plot styles allow datapoints to be plotted with errorbars indicating the uncertainties in either their vertical or horizontal positions:

- `yerrorbars`
- `xerrorbars`

Both of these take three columns of input data on 2D plots (or four on 3D plots). The first two (or three) of these represent the x-, y- (and z-) coordinates of the central position of each errorbar, while the last represents the uncertainty in either the x- and y-coordinate. The plot style `errorbars` is an alias for `yerrorbars`. Additionally, the following plot style allows datapoints to be plotted with both horizontal and vertical errorbars:

- `xyerrorbars`

This plot style takes four (or five) columns of data as input, the first two (or three) of which represent the x-, y- (and z-) coordinates of the central position of each errorbar. The last but one column gives the uncertainty in the x-coordinate, and the last column gives the uncertainty in the y-coordinate.

Each of the plot styles listed above has a corresponding partner which takes minimum and maximum limits for each errorbar, equivalent to writing 5^{+2}_{-3}, in place of a single symmetric uncertainty:

- `xerrorrange` – takes four (or five) columns of data.

- `yerrorrange` – takes four (or five) columns of data.
- `xyerrorrange` – takes six (or seven) columns of data.

The plot style `errorrange` is an alias of `yerrorrange`.

Corresponding plot styles also exist to plot data with errorbars along the z-axes of three-dimensional plots: `zerrorbars`, `zerrorrange`, `xzerrorbars`, `xzerrorrange`, `yzerrorbars`, `yzerrorrange`, `xyzerrorbars`, `xyzerrorrange`. Though it does not make sense to use these on two-dimensional plots, it is not an error to do so; they expect the same number of columns of input data on both two- and three-dimensional plots.

8.2.3 Shaded regions

The following plot styles allow regions of graphs to be shaded with color:

- `yerrorshaded`
- `shadedregion`

Both fill specified regions of graphs with the selected `fillcolor` and draw a line around the boundary of the region with the selected `color`, `linetype` and `linewidth`.

They differ in the format in which they expect the input data to be arranged. The `yerrorshaded` plot style is similar to the `yerrorrange` plot style: it expects three columns of data, specifying the x-coordinate and the minimum and maximum extremes of the vertical errorbar on each data point. The region contained between the upper and lower limits of these error bars is filled with color. Note that the data points must be sorted in order of either increasing or decreasing x-coordinate for sensible behaviour.

The `shadedregion` plot style takes only two columns of input data, specifying the x- and y-coordinates of a series of data points which are to be joined in a join-the-dots fashion. At the end of each dataset, the drawn path is closed and filled.

The use of these plot styles on three-dimensional graphs may produce unexpected results.

8.2.4 Barcharts and histograms

The following plot styles allow barcharts to be produced:

- `boxes`
- `impulses`
- `wboxes`

These styles differ in where the horizontal interfaces between the bars are placed along the abscissa axis and how wide the bars are. In the `boxes` plot style, the interfaces between the bars are at the midpoints between the specified data points by default (see, for example, Figure 8.1a). Alternatively, the widths of the bars may be set using the `set boxwidth` command. In this case, all of the bars will be centered on their specified x-coordinates, and have total widths

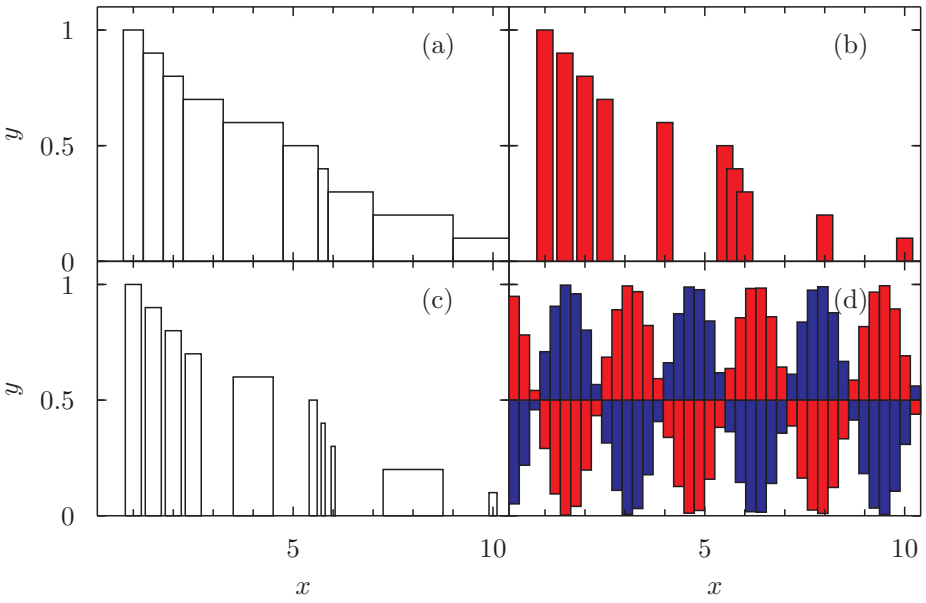


Figure 8.1: A gallery of the various bar chart styles which Pyxplot can produce. See the text for more details.

equalling that specified in the `set boxwidth` command. Consequently, there may be gaps between them, or they may overlap, as seen in Figure 8.1(b).

Having set a fixed box width, the default behaviour of scaling box widths automatically may be restored either with the `unset boxwidth` command, or by setting the boxwidth to a negative width.

In the `wboxes` plot style, the width of each bar is specified manually as an additional column of the input data file. This plot style expects three columns of data to be provided: the x- and y-coordinates of each bar in the first two, and the width of the bars in the third. Figure 8.1(c) shows an example of this plot style in use.

Finally, in the `impulses` plot style, the bars all have zero width; see Figure 8.2(c) for an example.

In all of these plot styles, the bars originate from the line $y = 0$ by default, as is normal for a histogram. However, should it be desired for the bars to start from a different vertical line, this may be achieved by using the `set boxfrom` command, for example:

```
set boxfrom 5
```

In this case, all of the bars would now originate from the line $y = 5$. Figure 8.2(b) shows the kind of effect that is achieved; for comparison, Figure 8.2(a) shows the same bar chart with the boxes starting from their default position of $y = 0$.

The bars may be filled using the `with fillcolor` modifier, followed by the name of a color:

```
plot 'data.dat' with boxes fillcolor blue
plot 'data.dat' with boxes fc 4
```

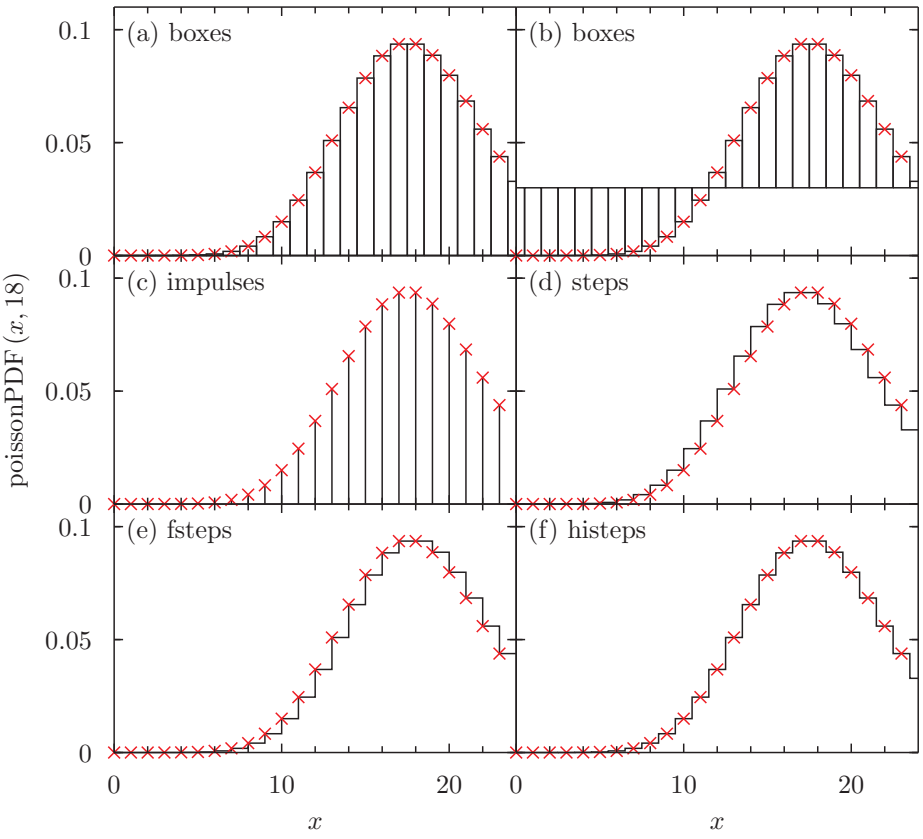


Figure 8.2: A second gallery of the various bar chart styles which Pyxplot can produce. See the text for more details. The script and data file used to produce this image are available on the Pyxplot website at http://www.pyxplot.org.uk/examples/Manual/03barchart1/.

Figures 8.1(b) and (d) demonstrate the use of filled bars.

The `boxes` and `wboxes` plot styles expect identically-formatted data when used on two- and three-dimensional plots; in the latter case, all bars are drawn in the plane $z = 0$. The `impulses` plot style takes an additional column of data on three-dimensional plots, specifying the z-coordinate at which each impulse should be drawn.

Stacked bar charts

If multiple data points are supplied to the `boxes` or `wboxes` plot styles at a common x-coordinate, then the bars are stacked one above another into a stacked barchart. Consider the following data file:

```
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 colors are being automatically selected from the palette, then a different palette color is used to plot the upper bar.

8.2.5 Steps

The following plot styles allow data to be plotted with a series of horizontal steps associated with each supplied data point:

- `steps`
- `fsteps`
- `histeps`

Each of these styles takes two columns of data, containing the x- and y-coordinates of each data point. They expect identically-formatted data regardless of whether they are used on two- and three-dimensional plots; in the latter case, the steps are drawn in the plane $z = 0$.

An example of their appearance is shown in Figures 8.2(d), (e) and (f); for clarity, the positions of each of the supplied data points are marked by red crosses.

These plot styles differ in their placement of the edges of each of the horizontal steps. The `steps` plot style places the right-most edge of each step on the data point it represents. The `fsteps` plot style places the left-most edge of each step on the data point it represents. The `histeps` plot style centers each step on the data point it represents.

8.2.6 Arrows

The following plot styles allow arrows or lines to be drawn on graphs with positions dictated by a series of data points:

- `arrows_head`

- `arrows_nohead`
- `arrows_twohead`

The plot style of `arrows` is an alias for `arrows_head`. Each of these plot styles take four columns of data on two-dimensional plots – x_1, y_1, x_2 and y_2 – or six columns of data on three-dimensional plots with additional z-coordinates. Each data point results in an arrow being drawn from the point (x_1, y_1, z_1) to the point (x_2, y_2, z_2). The three plot styles differ in the kinds of arrows that they draw: `arrows_head` draws an arrow head on each arrow at the point (x_2, y_2, z_2); `arrows_nohead` draws simple lines without arrow heads on either end; `arrows_twohead` draws arrow heads on both ends of each arrow.

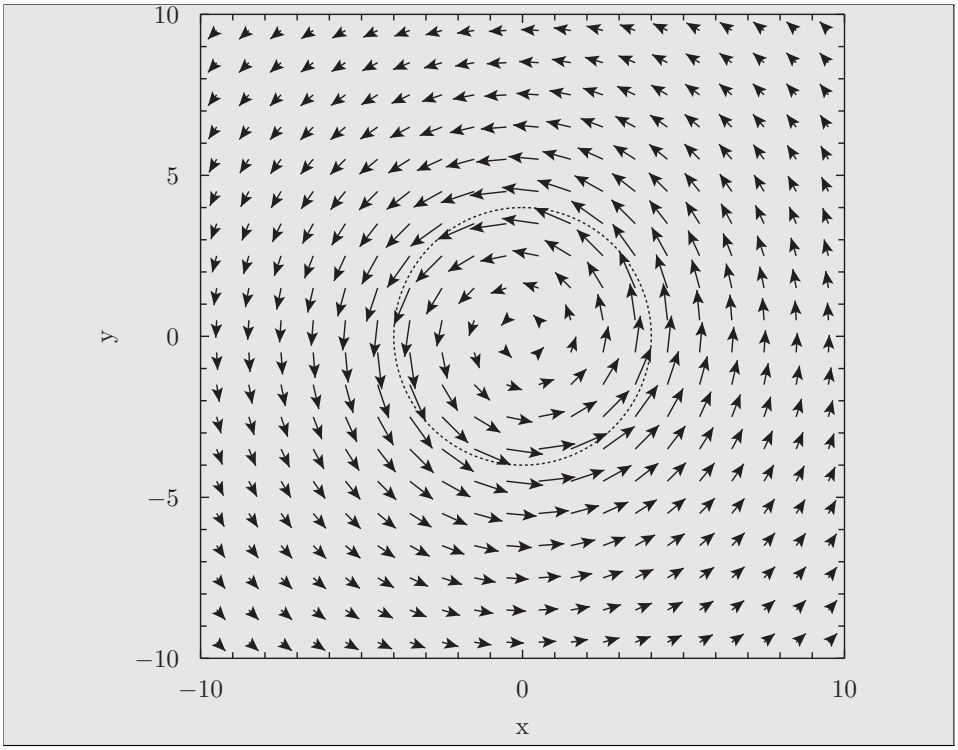
Example 16: A diagram of fluid flow around a vortex.

In this example we produce a velocity map of fluid circulating in a vortex. For simplicity, we assume that the fluid in the core of the vortex, at radii $r < 1$, is undergoing solid body rotation with velocity $v \propto r$, and that the fluid outside this core is behaving as a free vortex with velocity $v \propto 1/r$. First of all, we use a simple python script to generate a data file with the four columns:

```
from math import *
for i in range(-19,20,2):
 for j in range(-19,20,2):
  x = float(i)/2
  y = float(j)/2
  r = sqrt(x**2 + y**2) / 4
  theta = atan2(y,x)
  if (r < 1.0):  v = 1.3*r
  else :  v = 1.3/r
  vy = v * cos(theta)
  vx = v * -sin(theta)
  print "%7.3f %7.3f %7.3f %7.3f"%(x,y,vx,vy)
```

This data can then be plotted using the following Pyxplot script:

```
set size square
set width 9
set nokey
set xlabel 'x'
set ylabel 'y'
set trange [0:2*pi]
plot \
'vortex.dat' u 1:2:($1+$3):($2+$4) w arrows, \
parametric 4*sin(t):4*cos(t) w lt 2 col black
```



8.2.7 Color maps, contour maps and surface plots

The following plot styles differ from those above in that, regardless of whether a three-dimensional plot is being produced, they read in datapoints with x, y and z coordinates in three columns. The first two are useful for producing two-dimensional representations of (x, y, z) surfaces using colors or contours to show the magnitude of z, while the third is useful for producing three-dimensional graphs of such surfaces:

- colormap
- contourmap
- surface

They are discussed in detail in Sections 8.12, 8.13 and 8.14.1 respectively.

8.3 Labelling datapoints

The `label` modifier to the `plot` command may be used to add text labels next to datapoints, as in the following examples:

```
set samples 8
plot [2:5] x**2 label "$x=%.2f$"%($1) with points

plot 'datafile' using 1:2 label "%s"%($3)
```

Note that if a particular column of a data file contains strings which are to be used as labels, as in the second example above, syntax such as `"%s"%($3)` must be used to explicitly read the data as strings rather than as numerical quantities. As Pyxplot treats any whitespace as separating columns of data, such labels cannot contain spaces, though latex's $\sim$ character (a non-breaking space) can be used to achieve a space.

Data points can be labelled when plotted in any of the following plot styles: `arrows` (and similar styles), `dots`, `errorbars` (and similar styles), `errorrange` (and similar styles), `impulses`, `linespoints`, `lowerlimits`, `points`, `stars` and `upperlimits`. It is not possible to label datapoints plotted in other styles. Labels are rendered in the same color as the datapoints with which they are associated.

8.4 The `style` keyword

At times, the string of style keywords placed after the `with` modifier in `plot` commands can grow rather unwieldy in its length. For clarity, frequently used plot styles can be stored as numbered plot *styles*. The syntax for setting a numbered plot style is:

```
set style 2 points pointtype 3
```

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

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

8.5 Plotting functions in exotic styles

The use of plot styles which take more than two columns of input data to plot functions requires more than one function to be supplied. When functions are plotted with syntax such as

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

two columns of data are generated: the first contains values of x – plotted against the horizontal axis – and the second contains values of $\sin(x)$ – plotted against the vertical axis. Syntax such as

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

generates three columns of data. As before, the first contains values of x. The second and third contain samples from the colon-separated functions $f(x)$ and $g(x)$. Specifically, in this example, $g(x)$ provides the uncertainty in the value of $f(x)$. The `using` modifier may also be used in combination with such syntax, as in

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

though this example is not sensible. $g(x)$ would be plotted on the `y`-axis, against $f(x)$ on the `x`-axis. However, this is unlikely to be sensible because the range of values of x substituting into these expressions would correspond to the range of the plot's horizontal axis. The result might be particularly unexpected if the above were attempted with an autoscaling horizontal axis – Pyxplot would find itself autoscaling the `x`-axis range to the spread of values of $f(x)$, but find that this itself changed depending on the range of the `x`-axis. In this case, the user should have used the `parametric` plot option described in the next section.

8.6 Plotting parametric functions

Parametric functions are functions expressed in forms such as

$$\begin{aligned} x &= r\sin(t) \\ y &= r\cos(t), \end{aligned}$$

where separate expressions are supplied for the ordinate and abscissa values as a function of some free parameter t. The above example is a parametric representation of a circle of radius r. Before Pyxplot can usefully plot parametric functions, it is generally necessary to stipulate the range of values of t over which the function should be sampled. This may be done using the `set trange` command, as in the example

```
set trange [unit(0*rad):unit(2*pi*rad)]
```

or in the `plot` command itself. By default, values in the range $0 \leq t \leq 1$ are used. Note that the `set trange` command differs from other commands for setting axis ranges in that auto-scaling is not an allowed behaviour; an explicit range *must* be specified for t.

Having set an appropriate range for t, parametric functions may be plotted by placing the keyword `parametric` before the list of functions to be plotted, as in the following simple example which plots a circle:

```
set trange [unit(0*rev):unit(1*rev)]
plot parametric sin(t):cos(t)
```

Optionally, a range for t can be specified on a plot-by-plot basis immediately after the keyword `parametric`, and thus the effect above could also be achieved using:

```
plot parametric [unit(0*rev):unit(1*rev)] sin(t):cos(t)
```

The only difference between parametric function plotting and ordinary function plotting – other than the change of dummy variable from `x` to `t` – is that one fewer column of data is generated. Thus, whilst

```
plot f(x)
```

generates two columns of data, with values of x in the first column,

```
plot parametric f(t)
```

generates only one column of data.

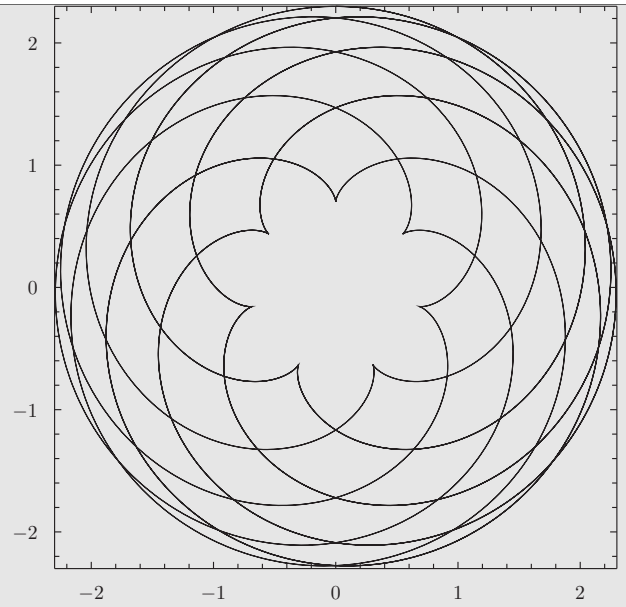
Example 17: Spirograph patterns.

Spirograph patterns are produced when a pen is tethered to the end of a rod which rotates at some angular speed ω_1 about the end of another rod, which is itself rotating at some angular speed ω_2 about a fixed central point. Spirographs are commonly implemented mechanically as wheels within wheels – epicycles within deferents, mathematically speaking – but in this example we implement them using the parametric functions

$$\begin{aligned} x &= r_1 \sin(t) + r_2 \sin(t r_1/r_2) \\ y &= r_1 \cos(t) + r_2 \cos(t r_1/r_2) \end{aligned}$$

which are simply the sum of two circular motions with angular velocities inversely proportional to their radii. The complexity of the resulting spirograph pattern depends on how rapidly the rods return to their starting configuration; if the two chosen angular speeds for the rods have a large lowest common multiple, then a highly complicated pattern will result. In the example below, we pick a ratio of $8 : 15$:

```
set nogrid
set nokey

r1 = 1.5
r2 = 0.8
set size square
set trange[0:40*pi]
set samples 2500
plot parametric r1*sin(t) + r2*sin(t*(r1/r2)) : \
r1*cos(t) + r2*cos(t*(r1/r2))
```



Other ratios of `r1`:`r2` such as $7:19$ and $5:19$ also produce intricate patterns.

8.6.1 Two-dimensional parametric surfaces

Pyxplot can also plot datasets which can be parameterised in terms of two free parameters u and v. This is most often useful in conjunction with the `surface` plot style, allowing any (u, v)-surface to be plotted (see Section 8.14.1 for details of the `surface` plot style). However, it also works in conjunction with any other plot style, allowing, for example, (u, v)-grids of points to be constructed.

As in the case of parametric lines above, the range of values that each free parameter should take must be specified. This can be done using the `set urange` and `set vrange` commands. These commands also act to switch Pyxplot between one- and two-dimensional parametric function evaluation; whilst the `set trange` command indicates that the next parametric function should be evaluated along a single raster of values of t, the `set urange` and `set vrange` commands indicate that a grid of (u, v) values should be used. By default, the range of values used for u and v is $0 \to 1$.

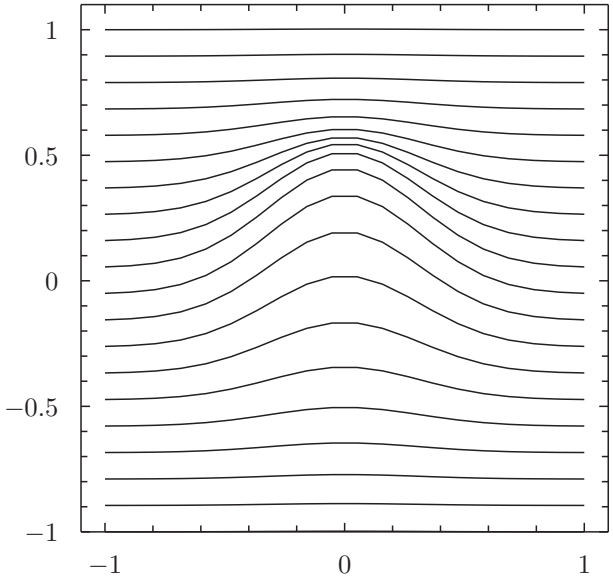
The number of samples to be taken can be specified using a command of the form

```
set sample grid 20x50
```

which specifies that 20 different values of u and 50 different values of v should be used, yielding a total of 1000 datapoints. The following example uses the `lines` plot style to generate a sequence of cross-sections through a two-dimensional Gaussian surface:

```
set num err quiet
set nokey
```

```
set size 7 square
set sample grid 20x20
set urange [-1:1] ; set vrange [-1:1]
set xrange [-1.1:1.1]
f(u,v) = 0.4*exp(-(u**2+v**2)/0.2)
plot parametric u:v+f(u,v) with l
```



The ranges of values to use for u and v may alternatively be specified on a dataset-by-dataset basis within the plot command, as in the example

```
plot parametric [0:1][0:1] u:v , \
     parametric [0:1] sin(t):cos(t)
```

Example 18: A three-dimensional view of a torus.

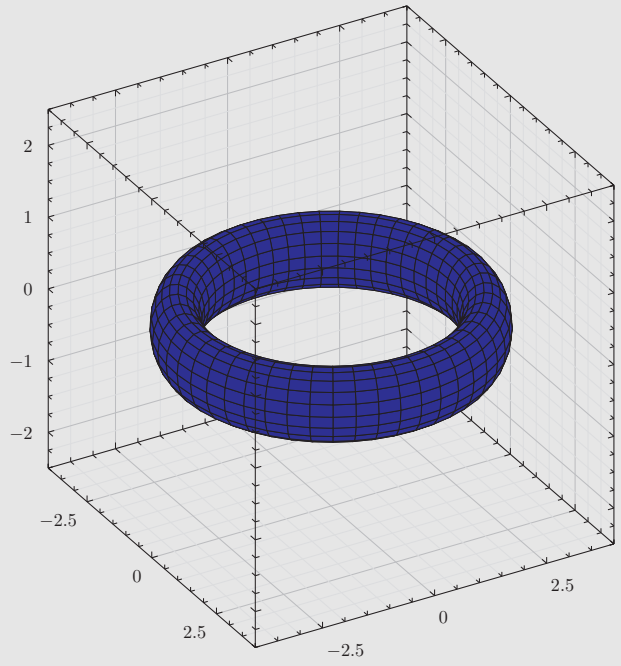
In this example we plot a torus, which can be parametrised in terms of two free parameters u and v as

$$\begin{aligned} x &= (R + r\cos(v))\cos(u) \\ y &= (R + r\cos(v))\sin(u) \\ z &= r\sin(v), \end{aligned}$$

where u and v both run in the range $[0 : 2\pi]$, R is the distance of the tube's center from the center of the torus, and r is the radius of the tube.

```
R = 3
r = 0.5
f(u,v) = (R+r*cos(v))*cos(u)
g(u,v) = (R+r*cos(v))*sin(u)
h(u,v) = r*sin(v)
```

```
set urange [0:2*pi]
set vrange [0:2*pi]
set zrange [-2.5:2.5]

set nokey
set size 8 square
set grid
set sample grid 50x20
plot 3d parametric f(u,v):g(u,v):h(u,v) with surf fillcol blue
```



Example 19: A three-dimensional view of a trefoil knot.

In this example we plot a trefoil knot, which is the simplest non-trivial knot in topology. Simply put, this means that it is not possible to untie the knot without cutting it. The knot follows the line

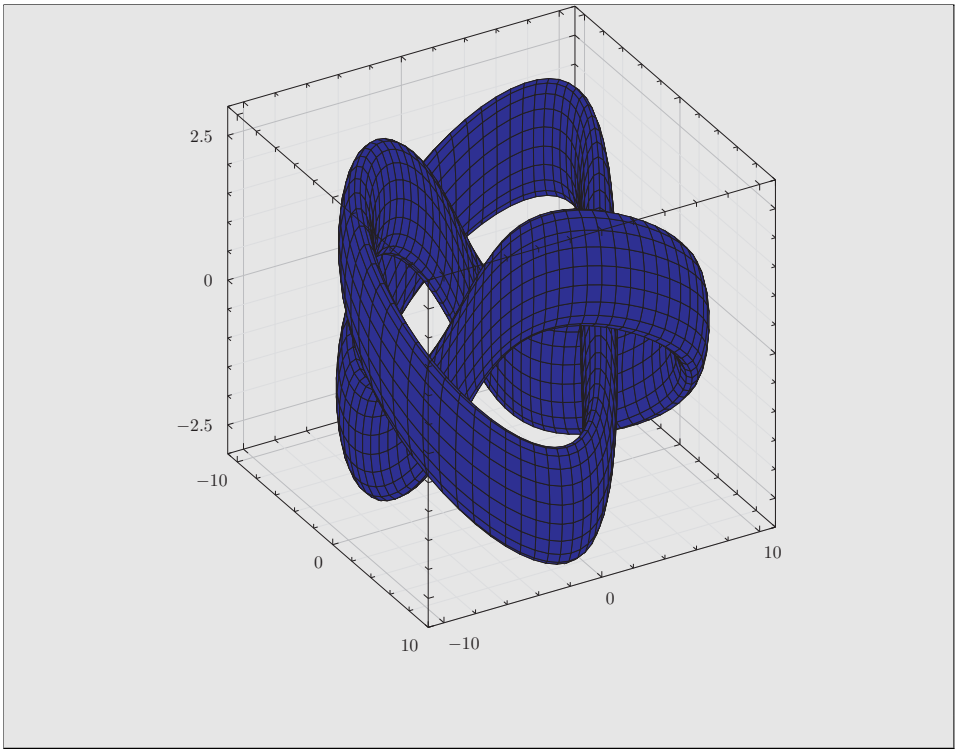
$$\begin{aligned} x &= (2+\cos(3t))\cos(2t) \\ y &= (2+\cos(3t))\sin(2t) \\ z &= \sin(3t), \end{aligned}$$

but in this example we construct a tube around this line using the following parameterisation:

$$\begin{aligned} x &= \cos(2u)\cos(v) + r\cos(2u)(1.5+\sin(3u)/2) \\ y &= \sin(2u)\cos(v) + r\sin(2u)(1.5+\sin(3u)/2) \\ z &= \sin(v) + R\cos(3u), \end{aligned}$$

where u and v run in the ranges $[0:2\pi]$ and $[-\pi:\pi]$ respectively, and r and R determine the size and thickness of the knot as in an analogous fashion to the previous example.

```
r = 5
R = 2
f(u,v) = cos(2*u)*cos(v) + r*cos(2*u)*(1.5+sin(3*u)/2)
g(u,v) = sin(2*u)*cos(v) + r*sin(2*u)*(1.5+sin(3*u)/2)
h(u,v) = sin(v)+R*cos(3*u)

set urange [0:2*pi]
set vrange [-pi:pi]

set nokey
set size 8 square
set grid
set sample grid 150x20
plot 3d parametric f(u,v):g(u,v):h(u,v) with surf fillcol blue
```



8.7 Graph legends

By default, plots are displayed with legends in their top-right corners. The textual description of each dataset is auto-generated 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 in the following examples:

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

In the latter case a blank title is specified, which indicates to Pyxplot that no entry should be made for the dataset in the legend. This allows for legends which contain only a subset of the datasets on a plot. Alternatively, the production of the legend can be completely turned off for all datasets using the command `set nokey`. Having issued this command, the production of keys can be resumed using the `set key` command.

The `set key` command can also be used to dictate how legends should be positioned on graphs, using a syntax along the lines of:

```
set key top right
```

The following recognised positional keywords are self-explanatory: `top`, `bottom`, `left`, `right`, `xcenter` and `ycenter`. Any single instance of the `set key` command can be followed by one horizontal alignment keyword and one vertical

alignment keyword; these keywords also affect the justification of the legend – for example, the keyword `left` aligns the legend with its left edge against the left edge of the plot.

Alternatively, the position of the legend can be indicated using one of the keywords `outside`, `below` or `above`. These cannot be combined with the horizontal and vertical alignment keywords above, and are used to indicate that the legend should be placed, respectively, outside the plot on its right side, centered beneath the plot, and centered above the plot.

Two comma-separated positional offset coordinates may be specified following any of the named positions listed above to fine-tune the position of the legend – the first value is assumed to be a horizontal offset and the second a vertical offset. Either may have units of length, or, if they are dimensionless, are assumed to be measured in centimeters, as the following examples demonstrate:

```
set key bottom left 0.0 -2
set key top xcenter 2*unit(mm),2*unit(mm)
```

By default, entries in the legend are automatically sorted into an appropriate number of columns. The number of columns to be used, can, instead, be stipulated using the `set keycolumns` command. This should be followed by either the integer number of desired columns, or by the keyword `auto` to indicate that the default behaviour of automatic formatting should be resumed:

```
set keycolumns 2
set keycolumns auto
```

8.8 Configuring axes

8.8.1 Adding additional axes

By default, plots have only one horizontal `x`-axis and one vertical `y`-axis. Additional axes may be added parallel to these and are referred to as, for example, the `x2` axis, the `x3` axis, and so forth up to a maximum of `x127`. In keeping with this nomenclature, the first axis in each direction can be referred to interchangeably as, for example, `x` or `x1`, or as `y` or `y1`. Further axes are automatically generated when statements such as the following are issued:

```
set x2label 'A second horizontal axis'
```

Such axes may alternatively be created explicitly using the `set axis` command, as in the example

```
set axis x3
```

or removed explicitly using the `unset axis` command, as in the example

```
unset axis x3
```

In either case, multiple axes can be created or removed in a single statement, as in the examples

```
unset axis x3x5x6 y2
set axis x2y2
```

The first axes `x1` and `y1` – and `z1` on three-dimensional plots – are unique in that they cannot be removed; all plots must have at least one axis in each perpendicular direction. Thus, the command `unset axis x1` does not remove this first axis, but merely 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
```

If an `x3`-axis did not previously exist, then the first will have implicitly created one. This would need to be removed with the `unset axis x3` command if it was not desired.

8.8.2 Selecting which axes to plot against

The axes against which data are plotted can be selected by passing the `axes` modifier to the `plot` command. By default, data is plotted against the first horizontal axis and the first vertical axis. In the following `plot` command the second horizontal axis and the third vertical axis would be used:

```
plot f(x) axes x2y3
```

It is also possible to plot data using a vertical axis as the abscissa axis using syntax such as:

```
plot f(x) axes y3x2
```

Similar syntax is used when plotting three-dimensional graphs, except that three perpendicular axes should be specified.

8.8.3 Plotting quantities with physical units

When data with non-dimensionless physical units are plotted against an axis, for example using any of the statements

```
plot [0:10] x*unit(m)
plot [0:10] x using 1:$2*unit(m)
plot [0*unit(m):1*unit(m)] x**2

set unit angle nodimensionless ; plot [0:1] asin(x)
```

the axis is set to share the particular physical dimensions of that unit, and thereafter no data with any other physical dimensions may be plotted against that axis. When the axis comes to be drawn, Pyxplot makes a decision about which physical unit should be used to label the axis. For example, in the default SI system and with no preferred unit of length set, axes with units of length might be displayed in millimeters, meters or kilometers depending on their scales.

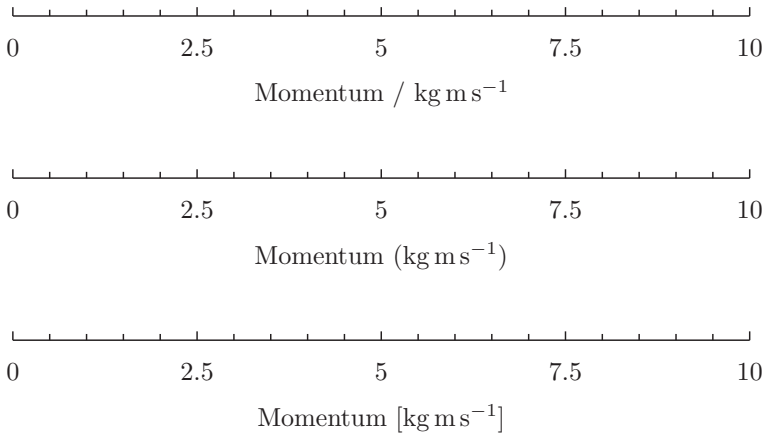
The chosen unit is indicated in one of three styles in the axis label, selected using the `set axisunitstyle` command:

```
set axisunitstyle ratio
set axisunitstyle bracketed
set axisunitstyle squarebracketed
```

The effect of these three options, respectively, is shown below for an axis with units of momentum. In each case, the axis label was set simply using

```
set xlabel "Momentum"
```

and the subsequent text was appended automatically by Pyxplot:



When the `set xformat` command is used (see Section 8.8.8), no indication of the units associated with axes are appended to axis labels, as the `set xformat` command can be used to hard-code this information. The user must include this information in the axis label manually if it is needed.

8.8.4 Specifying the positioning of axes

By default, the `x1`-axis is placed along the bottom of graphs and the `y1`-axis is placed up the left-hand side of graphs. On three-dimensional plots, the `z1`-axis is placed at the front of the graph. The second set of axes are placed opposite the first: the `x2`-, `y2`- and `z2`-axes are placed respectively along the top, right and back sides of graphs. Higher-numbered axes are placed alongside the `x1`-, `y1`- and `z1`-axes.

However, the position of any axis can be explicitly set using syntax of the form:

```
set axis x top
set axis y right
set axis z back
```

Horizontal axes can be set to appear either at the `top` or `bottom`; vertical axes can be set to appear either at the `left` or `right`; and z-axes can be set to appear either at the `front` or `back`.

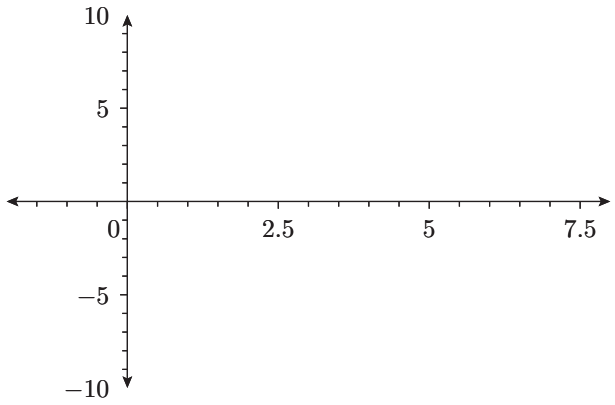
8.8.5 Configuring the appearance of axes

The `set axis` command also accepts the following keywords alongside the positional keywords listed above, which specify how the axis should appear:

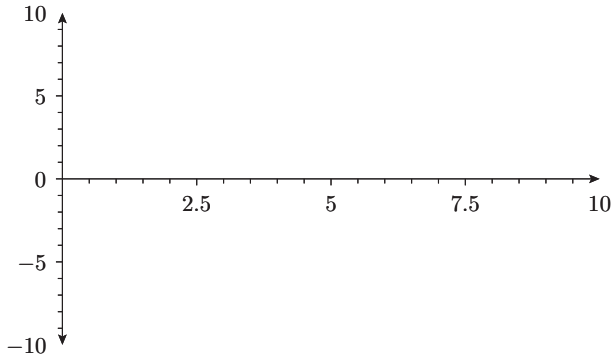
- `arrow` – Specifies that an arrowhead should be drawn on the right/top end of the axis. [**Not default**].
- `atzero` – Specifies that rather than being placed along an edge of the plot, the axis should mark the lines where the perpendicular axes `x1`, `y1` and/or `z1` are zero. [**Not default**].
- `automirrored` – Specifies that an automatic decision should be made between the behaviour of `mirrored` and `nomirrored`. If there are no axes on the opposite side of the graph, a mirror axis is produced. If there are already axes on the opposite side of the graph, no mirror axis is produced. [**Default**].
- `fullmirrored` – Similar to `mirrored`. Specifies that this axis should have a corresponding twin placed on the opposite side of the graph with mirroring ticks and labelling. [**Not default**; see `automirrored`].
- `invisible` – Specifies that the axis should not be drawn; data can still be plotted against it, but the axis is unseen. See Example 24 for a plot where all of the axes are invisible.
- `linked` – Specifies that the axis should be linked to another axis; see Section 8.8.9.
- `mirrored` – Specifies that this axis should have a corresponding twin placed on the opposite side of the graph with mirroring ticks but with no labels on the ticks. [**Not default**; see `automirrored`].
- `noarrow` – Specifies that no arrowheads should be drawn on the ends of the axis. [**Default**].
- `nomirrored` – Specifies that this axis should not have any corresponding twins. [**Not default**; see `automirrored`].
- `notatzero` – Opposite of `atzero`; the axis should be placed along an edge of the plot. [**Default**].
- `notlinked` – Specifies that the axis should no longer be linked to any other; see Section 8.8.9. [**Default**].
- `reversearrow` – Specifies that an arrowhead should be drawn on the left/bottom end of the axis. [**Not default**].
- `twowayarrow` – Specifies that arrowheads should be drawn on both ends of the axis. [**Not default**].
- `visible` – Specifies that the axis should be displayed; opposite of `invisible`. [**Default**].

The following simple examples demonstrate the use of some of these configuration options:

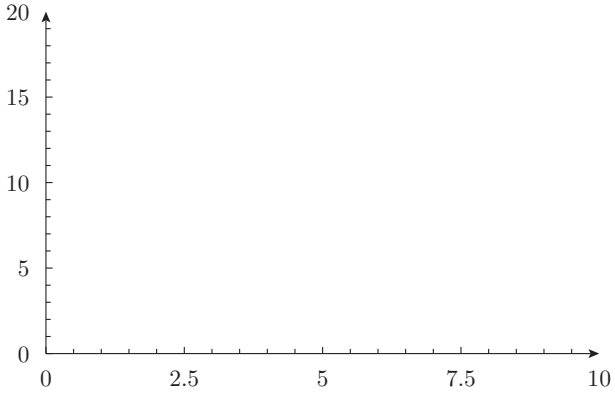
```
set axis x atzero twoway
set axis y atzero twoway
plot [-2:8][-10:10]
```




```
set axis x atzero arrow
set axis y atzero twoway
plot [0:10][-10:10]
```




```
set axis x notatzero arrow nomirror
set axis y notatzero arrow nomirror
plot [0:10][0:20]
```



8.8.6 Setting the color of axes

The colors of axes can be controlled via the `set axescolor`. The following example would set axes to be drawn in blue:

```
set axescolor blue
```

Any of the color names listed in Section 19.4 can be used, as can any object of type `color`, e.g. `rgb(0.2,0.1,0.8)`.

8.8.7 Specifying where ticks should appear along axes

By default, Pyxplot places a series of tick marks at significant points along each axis, with the most significant points being labelled. Labelled tick marks are termed *major* ticks, and unlabelled tick marks are termed *minor* ticks. The position and appearance of the major ticks along the `x`-axis can be configured using the `set xtics` command, which has the following syntax:

```
set xtics
    [ ( axis | border | inward | outward | both ) ]
    [ ( autofreq
          | [<minimum>,] <increment> [, <maximum>]
          | \( { '<label>' <position> } \)
        ] )
```

The corresponding `set mxtics` command, which has the same syntax as above, configures the appearance of the minor ticks along the `x`-axis. Analogous commands such as `set ytics` and `set mx2tics` configure the major and minor ticks along other axes.

The keywords `inward`, `outward` and `both` are used to configure how the ticks appear – whether they point inward, towards the plot, as is default, or outwards towards the axis labels, or in both directions. The keyword `axis` is an alias for `inward`, and `border` an alias for `outward`.

The remaining options are used to configure where along the axis ticks are placed. If a series of comma-separated values `<minimum>, <increment>, <maximum>` are specified, then ticks are placed at evenly spaced intervals between the specified limits. The `<minimum>` and `<maximum>` values are optional; if only one value is specified then it is taken to be the step size between ticks. If two values are specified, then the first is taken to be `<minimum>`. In the case of logarithmic axes, `<increment>` is applied as a multiplicative step size, and should be dimensionless. For example:

```
set xtics 0,1,10 # Ticks at 0,1,2,...,10
set log x
set xtics 2,2  # Ticks at 2,4,8,16 ...
```

Alternatively, if a bracketed list of quoted tick labels and tick positions are provided, then ticks can be placed on an axis manually and each given its own textual label. The quoted tick labels may be omitted, in which case they are automatically generated:

```
set xtics ("a" 1, "b" 2, "c" 3)
set xtics (1,2,3)
```

The keyword `autofreq` overrides any manual selection of ticks which may have been placed on an axis and resumes the automatic placement of ticks along it. The `show xtics` command, together with its companions such as `show x2tics`

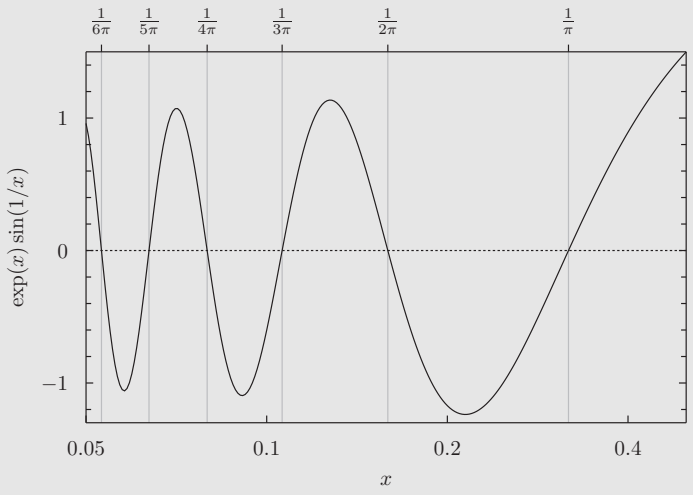
and `show ytics`, may be used to query the current ticking options. The `set noxtics` command may be used to stipulate that no ticks should appear along a particular axis:

```
set noxtics
show xtics
```

Example 20: A plot of the function $\exp(x)\sin(1/x)$.

In this example we produce a plot illustrating some of the zeroes of the function $\exp(x)\sin(1/x)$. We set the `x`-axis to have tick marks at $x = 0.05$, 0.1, 0.2 and 0.4. The `x2`-axis has custom labelled ticks at $x = 1/\pi, 2/\pi$, etc., pointing outwards from the plot. The left-hand `y`-axis has tick marks placed automatically whereas the `y2`-axis has no tics at all.

```
set log x1x2
set xrange [0.05:0.5]
set axis x2 top linked x
set xtics 0.05, 2, 0.4
set x2tics border \
(r"$\frac{1}{\pi}$" 1/pi,      r"$\frac{1}{2\pi}$" 1/(2*pi), \
r"$\frac{1}{3\pi}$" 1/(3*pi), r"$\frac{1}{4\pi}$" 1/(4*pi), \
r"$\frac{1}{5\pi}$" 1/(5*pi), r"$\frac{1}{6\pi}$" 1/(6*pi))
set grid x2
set nokey
set xlabel '$x$'
set ylabel '$\\exp(x)\sin(1/x)$'
plot exp(x)*sin(1/x), 0
```



8.8.8 Configuring how tick marks are labelled

By default, the major tick marks along axes are labelled with representations of the values represented at each point, each accurate to the number of significant

figures specified using the `set numerics sigfig` command. These labels may appear as decimals, such as 3.142, in scientific notion, as in 3×10^8, or, on logarithmic axes where a base has been specified for the logarithms, using syntax such as[1]

```
set log x1 2
```

in a format such as 1.5×2^8.

The `set xformat` command – together with its companions such as `set yformat`[2] – is used to manually specify an explicit format for the axis labels to take, as demonstrated by the following pair of examples:

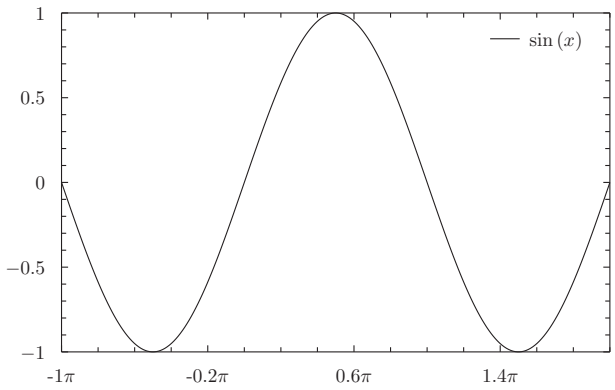
```
set xformat "%.2f"%(x)
set yformat "%s$^\prime$"%(y/unit(feet))
```

The first example specifies that ordinate values should be displayed to two decimal places along the `x`-axis; the second specifies that distances should be displayed in feet along the `y`-axis. Note that the dummy variable used to represent the ordinate value is `x`, `y` or `z` depending on the direction of the axis, but that the dummy variable used in the `set x2format` command is still `x`. The following pair of examples both have the equivalent effect of returning the `x2`-axis to its default system of tick labels:

```
set x2format auto
set x2format "%s"%(x)
```

The following example specifies that ordinate values should be displayed as multiples of π:

```
set xformat "%s$\pi$"%(x/pi)
plot [-pi:2*pi] sin(x)
```



Note that where possible, Pyxplot intelligently changes the positions along axes where it places the ticks to reflect significant points in the chosen labelling system. The extent to which this is possible depends on the format string supplied. It is generally easier when continuous-varying numerical values are

[1] Note that the `x` axis must be referred to as `x1` here to distinguish this statement from `set log x2`.

[2] There is no `set mxformat` command since minor axis ticks are never labelled unless labels are explicitly provided for them using the syntax `set mxtics (...)`.

substituted into strings, rather than discretely-varying values or strings. Thus, rather than

✗
```
set xformat "%d"%(floor(x))
```

the following is preferred

✔
```
set xformat "%d"%(x)
```

and rather than

✗
```
set xformat "%s"%date.str()
```

the following is preferred

✔
```
set xformat "%d/%02d/%d"%(date.toDayOfMonth(), \
date.toMonthNum(), date.toYear())
```

Changing the slant of axis labels

The `set xformat` command and its companions may also be followed by keywords which control the angle at which tick labels are drawn. By default, all tick labels are written horizontally, a behaviour which may be reproduced by issuing the command:

```
set xformat auto horizontal
```

Alternatively, tick labels may be set to be written vertically, by issuing the command

```
set xformat auto vertical
```

or to be written at any clockwise rotation angle from the horizontal using commands of the form

```
set xformat auto rotate 10
```

Axis labels may also be made to appear at arbitrary rotations using commands such as

```
set unit angle nodimensionless
set xlabel "I'm upside down" rotate unit(0.5*revolution)
```

Removing axis tick labels

Axes may be set to have no textual labels associated with the ticks along them using the command:

```
set xformat ""
```

This is particularly useful when compiling galleries of plots using linked axes (see the next section) and the multiplot environment (see Chapter 10).

8.8.9 Linked axes

Often it may be desired that multiple axes on a graph share a common range, or be related to one another by some algebraic expression. For example, a plot with wavelength λ of light as one axis may usefully also have parallel axes showing frequency of light $\nu = c/\lambda$ or photon energy $E = hc/\lambda$. The following example sets the `x2` axis to share a common range with the `x` axis:

```
set axis x2 linked x
```

An algebraic relationship between two axes may be set by stating the algebraic relationship after the keyword `using`, as in the following example which implement the formulae shown above for the frequency and energy of photons of light as a function of their wavelength:

```
set xrange [200*unit(nm):unit(800*nm)]
set axis x2 linked x1 using phy.c/x
set axis x3 linked x2 using phy.h*x
```

As in the `set xformat` command, a dummy variable of `x`, `y` or `z` is used in the linkage expression depending on the direction of the axis being linked to, but a dummy variable of `x` is still used when linking to, for example, the `x2` axis.

As these examples demonstrate, the functions used to link axes need not be linear. In fact, axes with any arbitrary mapping between position and value can be produced by linked in a non-linear fashion to another linear axis, which, if desired, can then be hidden using the `set axis invisible` command. Multi-valued mappings are also permitted. Any data plotted against the following `x2`-axis for a suitable range of `x`-axis

```
set axis x2 linked x1 using x**2
```

would appear twice, symmetrically on either side of $x = 0$.

Insofar as is possible, linked axes autoscale intelligently when no range is set. Thus, if the `x2`-axis is linked to the `x`-axis, and no range to set for the `x`-axis, the `x`-axis will autoscale to include all of the data plotted against both itself and the `x2`-axis. Similarly, if the `x2`-axis is linked to the `x`-axis by means of some algebraic expression, the `x`-axis will attempt to autoscale to include the data plotted against the `x2`-axis, though in some cases – especially with non-monotonic linking functions – this may prove too difficult. Where Pyxplot detects that it has failed, a warning is issued recommending that a hard range be set for – in this example – the `x`-axis.

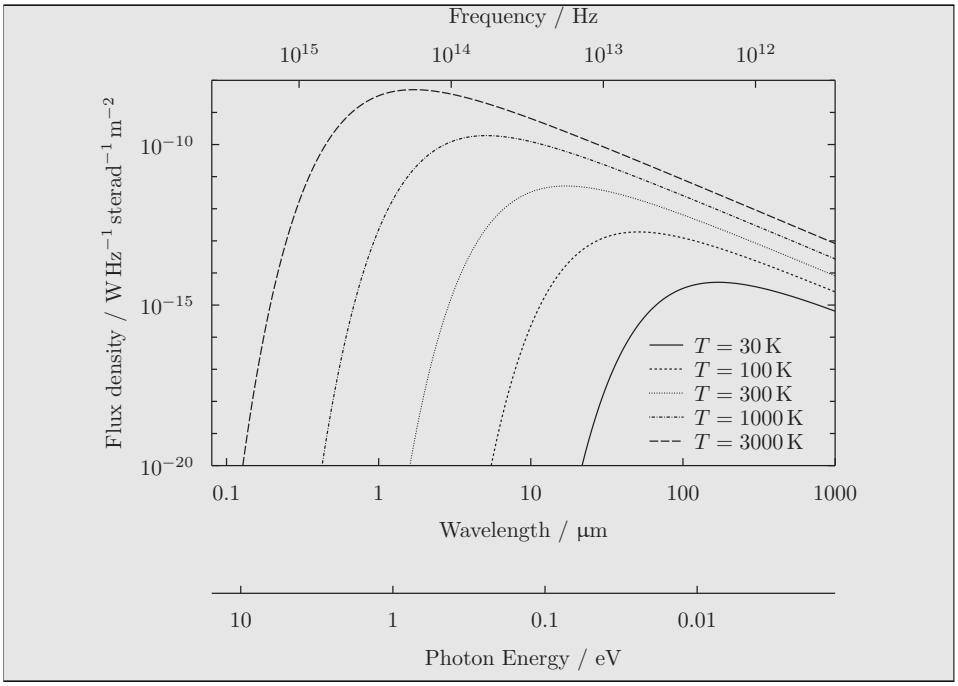
Example 21: A plot of many blackbodies demonstrating the use of linked axes.

In this example we produce a plot of blackbody spectra for five different temperatures T, using the Planck formula

$$B_\nu(\nu, T) = \left(\frac{2h^3}{c^2}\right) \frac{\nu^3}{\exp(h\nu/kT) - 1}$$

which is evaluated in Pyxplot by the system-defined mathematical function `Bv(nu,T)`. We use the axis linkage commands listed as an example in the text of Section 8.8.9 to produce three parallel horizontal axes showing wavelength of light, frequency of light and photon energy.

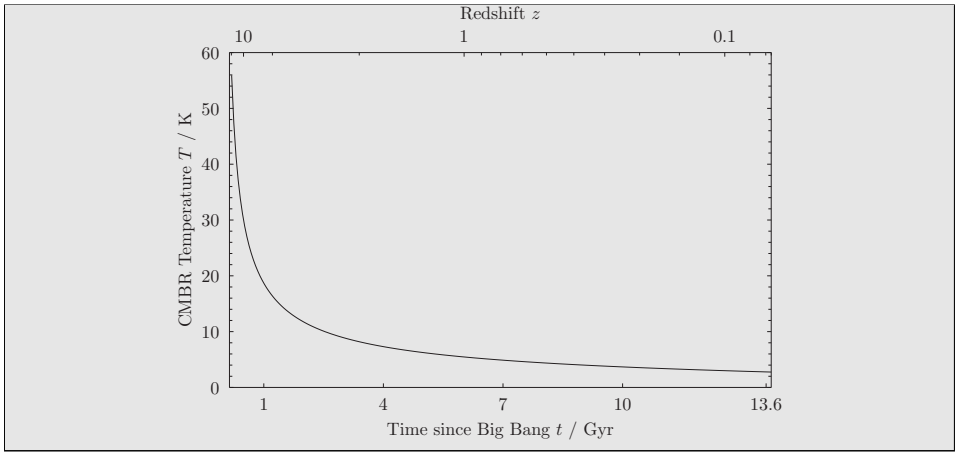
```
set numeric display latex
set unit angle nodimensionless
set log x y
set key bottom right
set ylabel  "Flux density" ; set unit preferred W/Hz/m**2/sterad
set x1label "Wavelength"
set x2label "Frequency"     ; set unit of frequency Hz
set x3label "Photon Energy" ; set unit of energy eV
set axis x2 linked x1 using phy.c/x
set axis x3 linked x2 using phy.h*x
set xtics unit(0.1*um),10
set x2tics 1e12*unit(Hz),10
set x3tics 0.01*unit(eV),10
set xrange [80*unit(nm):unit(mm)]
set yrange [1e-20*unit(W/Hz/m**2/sterad):]

bb(wlen,T) = phy.Bv(phy.c/wlen,T)

plot bb(x,  30) title r"$T=  30$\,K", \
bb(x, 100) title r"$T= 100$\,K", \
bb(x, 300) title r"$T= 300$\,K", \
bb(x,1000) title r"$T=1000$\,K", \
bb(x,3000) title r"$T=3000$\,K"
```



Example 22: A plot of the temperature of the CMBR as a function of redshift demonstrating non-linear axis linkage.

In this example we produce a plot of the temperature of the cosmic microwave background radiation (CMBR) as a function of time t since the Big Bang, on the `x`-axis, and equivalently as a function of redshift z, on the `x2`-axis. The specialist cosmology function `ast_Lcdm_z(`t`,`H_0`,`Ω_M`,`Ω_Λ`)` is used to make the highly non-linear conversion between time t and redshift z, adopting some standard values for the cosmological parameters H_0, Ω_M and Ω_Λ. Because the temperature of the CMBR is most easily expressed as a function of redshift as $T = 2.73\,\mathrm{K}/(1+z)$, we plot this function against axis `x2`.

```
h0 = 70
omega_m = 0.27
omega_l = 0.73
age = ast.Lcdm_age(h0,omega_m,omega_l)
set xrange [0.01*age:0.99*age]
set xtics (unit(1*Gyr),unit(4*Gyr),unit(7*Gyr),unit(10*Gyr),unit(13.6*Gyr))
set unit of time Gyr
set axis x2 linked x using ast.Lcdm_z(age-x,h0,omega_m,omega_l)
set xlabel "Time since Big Bang $t$"
set ylabel "CMBR Temperature $T$"
set x2label "Redshift $z$"
plot unit(2.73*K)*(1+x) ax x2y1
```



8.9 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 default horizontal and vertical axes (which are the first axes in each direction unless set otherwise in the configuration file; see Chapter 19). 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:

```
set nogrid x2x3
```

The colors of gridlines can be controlled via the `set gridmajcolor` and `set gridmincolor` commands, which control the gridlines emanating from major and minor axis ticks respectively. The following example would set the minor grid lines on a graph to be drawn in blue:

```
set gridmajcolor gray70
set gridmincolor blue
```

Any of the color names listed in Section 19.4 can be used, as can any object of type `color`.

8.10 Clipping behaviour

The treatment of datapoints close to the edges of plots may be specified using the `set clip` command, which provides two options. Either datapoints close to the axes can be clipped and not allowed to overrun the axes – specified by `set clip` – or such datapoints may be allowed to extend over the lines of the axes – specified by `set noclip` and the default behaviour.

8.11 Labelling graphs

The `set arrow` and `set label` commands allow arrows and text labels to be added to graphs to label significant points or to add simple vector graphics to them.

8.11.1 Arrows

The `set arrow` command may be used to draw arrows on top of graphs; its syntax is illustrated by the following simple example:

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

Optionally, a third coordinate may be specified. On 2D plots, this is ignored. If no third coordinate is supplied then a value of $z = 0$ is substituted when the arrow is plotted on 3D graphs. The number `1` immediately following `set arrow` specifies an identification number for the arrow, allowing it to be subsequently removed via the command

```
unset arrow 1
```

or equivalently, via

```
set noarrow 1
```

or to be replaced with a different arrow by issuing a new command of the form `set arrow 1 ...`. The `set arrow` command may be followed by the keyword `with` to specify the style of the arrow. The keywords `nohead`, `head` and `twohead`, placed after the keyword `with`, can be used to generate arrows with no arrow heads, normal arrow heads, or with two arrow heads. `twoway` is an alias for `twohead`, as in the following example:

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

Line types, line widths and colors can also be specified after the keyword `with`, as in the example:

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

The coordinates for the start and end points of the arrow can be specified in a range of coordinate systems. The coordinate system to be used should be specified immediately before the coordinate value. The default system, `first` measures the graph using the `x`- and `y`-axes. The `second` system uses the `x2`-

and y2-axes. `axis<n>` specifies that the position is to be measured along the n th horizontal or vertical axis – for example, `axis3`. This allows the graph to be measured with reference to any arbitrary axis on plots which make use of large numbers of parallel axes (see Section 8.8.1). The `page` and `graph` systems both measure in centimeters 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  y0 \
            to   screen x1, screen y1 \
            with nohead
```

8.11.2 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 `at` keyword, can be specified in any of the coordinate systems described for arrows above. As above, either two or three coordinates may be supplied. 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
```

A color can also be specified, if desired, using the `with color` modifier. For example, the following would produce a green label at the origin:

```
set label 2 'This label is green' at 0, 0 with color green
```

The size of the text can be set using the `with fontsize` modifier:

```
set label 3 'A Big Blue Label' at 0,0 with col blue fontsize 4
```

Alternatively, it may be set globally using the `set fontsize` command. This applies not only to the `set label` command, but also to plot titles, axis labels, keys, etc. The value supplied should be a multiplicative factor greater than zero; a value of `2` would cause text to be rendered at twice its normal size, and a value of `0.5` would cause text to be rendered at half its normal size.

The `set textcolor` command can be used to globally set the color 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 color from the present palette, or by a color name. A list of the recognised color names can be found in Section 19.4. For example:

```
set textcolor 2
set textcolor 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`, `center` or `right`, and the latter takes the options `bottom`, `center` or `top`, for example:

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

Example 23: A diagram of the atomic lines of hydrogen.

The wavelengths of the spectral lines of atomic hydrogen are given by the Rydberg formula,

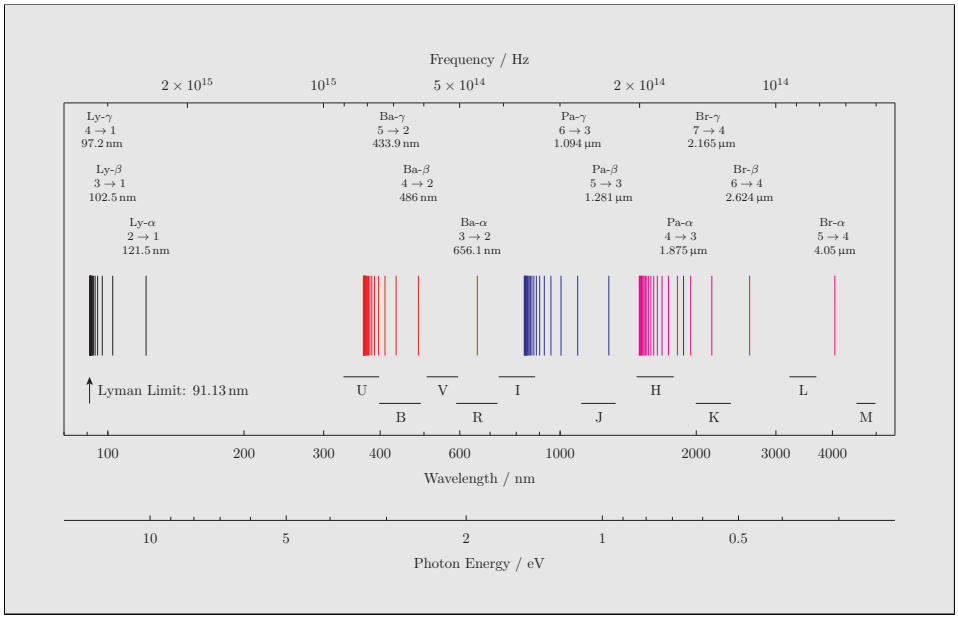
$$\frac{1}{\lambda} = R_{\mathrm{H}} \left(\frac{1}{n^2} - \frac{1}{m^2} \right),$$

where λ is wavelength, R_{H} is the Rydberg constant, predefined in Pyxplot as the variable `phy_Ry`, and `n` and `m` are positive non-zero integers such that `m>n`. The first few series are called the Lyman series (`n`= 1), the Balmer series (`n`= 2), the Paschen series (`n`= 3) and the Brackett series (`n`= 4). Within each series, the lines are given Greek letter designations – α for `m`=`n`+1, β for `m`=`n`+2, and so forth.

In the following example, we produce a diagram of the lines in the first four series, drawing the first 20 lines within each. At the bottom of the diagram, we overlay indications of the wavelengths of ten color filters commonly used by astronomers (data taken from Binney & Merrifield, *Galactic Astronomy*, Princeton, 1998).

```
set numeric display latex
set width 20
set size ratio 0.4
set numerics sf 4
set log x
set x1label "Wavelength"
set x2label "Frequency"     ; set unit of frequency Hz
set x3label "Photon Energy" ; set unit of energy eV
set axis x2 linked x1 using phy.c/x
set axis x3 linked x2 using phy.h*x
set noytics ; set nomytics

# Draw lines of first four series of hydrogen lines
an=2
n=1
foreach SeriesName in ["Ly","Ba","Pa","Br"]
{
for m=n+1 to n+21
{
wl = 1/(phy.Ry*(1/n**2-1/m**2))
set arrow an from wl,0.3 to wl,0.6 with nohead col n
if (m-n==1) { ; GreekLetter = r"\alpha" ; }
if (m-n==2) { ; GreekLetter = r"\beta"  ; }
if (m-n==3) { ; GreekLetter = r"\gamma" ; }
if (m-n<4)
{
set label an r"\parbox{5cm}{\footnotesize\center{\
%s-$%s$\newline $%d\to%d$\newline %s\newline}}" \
%(SeriesName,GreekLetter,m,n,wl) at wl,0.55+0.2*(m-n) \
hal center val center
}
an = an+1
}
n=n+1
}
```

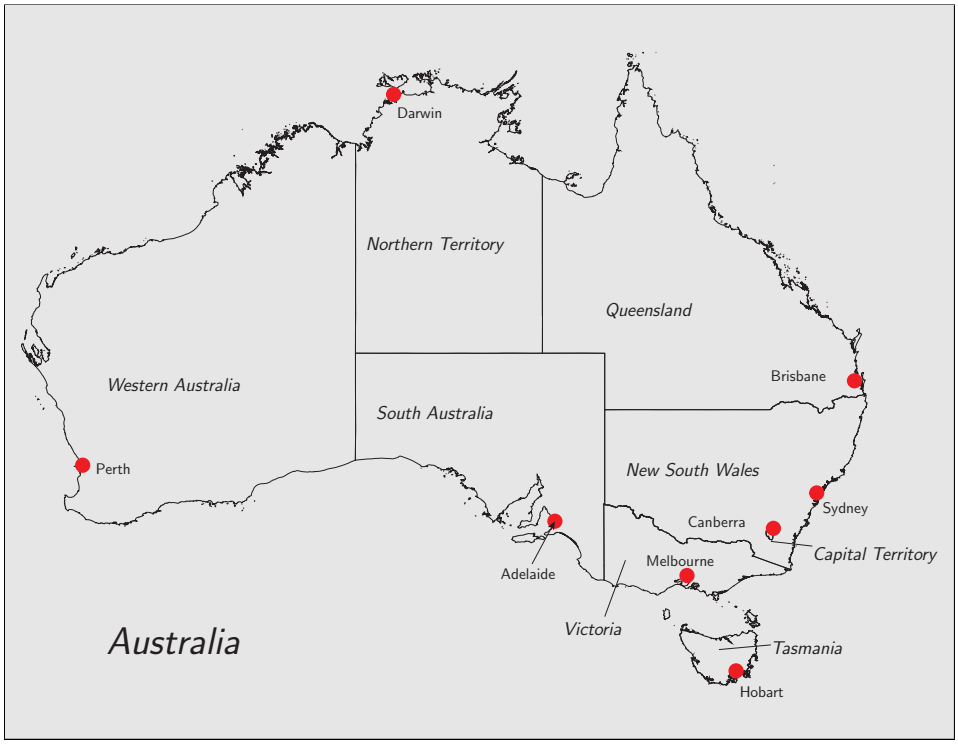
```
# Label astronomical photometric colors
foreach datum i,name,wl_c,wl_w in "--" using \
1:"%s"%($2):($3*unit(nm)):($4*unit(nm))
{
arry = 0.12+0.1*(i%2) # Vertical positions for arrows
laby = 0.07+0.1*(i%2) # Vertical positions for labels
x0 = (wl_c-wl_w/2) # Shortward end of passband
x1 =  wl_c         # Centre of passband
x2 = (wl_c+wl_w/2) # Longward end of passband
set arrow an from x0,arry to x2,arry with nohead
set label an name at x1,laby hal center val center
an = an+1
}
1 U  365   66
2 B  445   94
3 V  551   88
4 R  658  138
5 I  806  149
6 J 1220  213
7 H 1630  307
8 K 2190  390
9 L 3450  472
10 M 4750  460
END

# Draw a marker for the Lyman limit
ll = 91.1267*unit(nm)
set arrow 1 from ll,0.12 to ll,0.22
set label 1 "Lyman Limit: %s"%(ll) at 95*unit(nm),0.17 \
hal left val center

# Finally produce plot
plot [80*unit(nm):5500*unit(nm)][0:1.25]
```


Frequency / Hz
2 × 10^15
10^15
5 × 10^14
2 × 10^14
10^14
Ly-γ
4 → 1
97.2 nm
Ba-γ
5 → 2
433.9 nm
Pa-γ
6 → 3
1.094 µm
Br-γ
7 → 4
2.165 µm
Ly-β
3 → 1
102.5 nm
Ba-β
4 → 2
486 nm
Pa-β
5 → 3
1.281 µm
Br-β
6 → 4
2.624 µm
Ly-α
2 → 1
121.5 nm
Ba-α
3 → 2
656.1 nm
Pa-α
4 → 3
1.875 µm
Br-α
5 → 4
4.05 µm
Lyman Limit: 91.13 nm
U
B
V
R
I
J
H
K
L
M
100
200
300
400
600
1000
2000
3000
4000
Wavelength / nm
10
5
2
1
0.5
Photon Energy / eV

Example 24: A map of Australia.

In this example, we use Pyxplot to plot a map of Australia, using a coastal outline obtained from http://www.maproom.psu.edu/dcw/. We use the `set label` command to label the states and major cities. The files `ex_map_1.dat.gz` and `ex_map_2.dat` can be found in the Pyxplot installation tarball in the directory `doc/examples/`.

```
set size 20 ratio (45-10)/(154-112)*cos(25*unit(deg))
set xrange [112:154]
set yrange [-45:-10]

# We want a plot without axes or key
set nokey
set axis x invisible
set axis y invisible

# Labels for the states
set label 1 r'{\large \sf \slshape Western Australia}' 117, -28
set label 2 r'{\large \sf \slshape South Australia}' 130, -29.5
set label 3 r'{\large \sf \slshape Northern Territory}' 129.5, -20.5
set label 4 r'{\large \sf \slshape Queensland}' 141,-24
set label 5 r'{\large \sf \slshape New South Wales}' 142,-32.5
set label 6 r'{\large \sf \slshape Victoria}' 139,-41
set arrow 6 from 141,-40 to 142, -37 with nohead
set label 7 r'{\large \sf \slshape Tasmania}' 149,-42
set arrow 7 from 149, -41.5 to 146.5, -41.75 with nohead
set label 8 r'{\large \sf \slshape Capital Territory}' 151,-37
set arrow 8 from 151, -36.25 to 149, -36 with nohead

# Labels for the cities
set label 10 r'{\sf Perth}' 116.5, -32.4
set label 11 r'{\sf Adelaide}' 136, -38
set arrow 11 from 137.5,-37.2 to 138.601, -34.929
set label 12 r'{\sf Darwin}' 131, -13.5
set label 13 r'{\sf Brisbane}' 149, -27.5
set label 14 r'{\sf Sydney}' 151.5, -34.5
set label 15 r'{\sf Melbourne}' 143, -37.3
set label 16 r'{\sf Hobart}' 147.5, -44.25
set label 17 r'{\sf Canberra}' 145, -35.25

# A big label saying "Australia"
set label 20 r'{\Huge \sf \slshape Australia}' 117,-42

# Plot the coastline and cities
plot 'map_1.dat.gz' every ::1 with lines, \
'map_2.dat' with points pointtype 17 pointsize 2
```



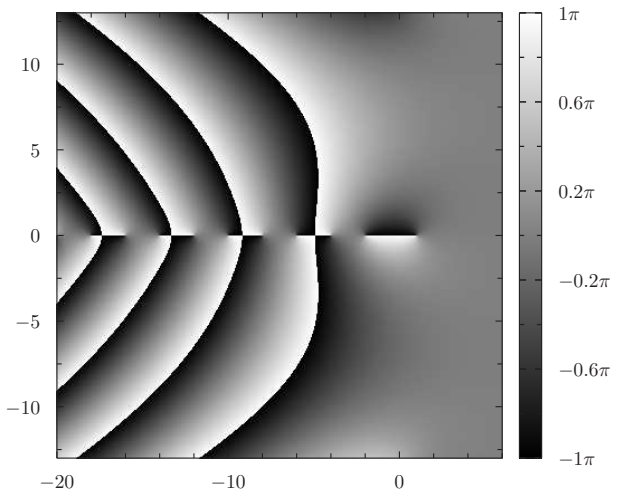
8.12 Color maps

Color maps provide a graphical means of producing two-dimensional representations of (x, y, z) surfaces, or equivalently of producing maps of the values $z(x, y)$ of functions of two variables. Each point in the (x, y) plane is assigned a color which indicates the value z associated with that point. In this section, we refer to the third coordinate as c_1 rather than z, to distinguish it from the third axes of three-dimensional plots[3].

In the following simple example, a color map of the complex argument of the Riemann zeta function $\zeta(z)$ is produced, taking the (x, y) plane to be an Argand plane, with x being the real axis, and y being the imaginary axis. Each point in the plane has an associated value of c_1.

```
set numerics complex
set nokey
set size 8 square
set samples grid 400x400
set c1range[-pi:pi]
set c1format r"$%s\pi$"%(c/pi)
plot [-20:6][-13:13] arg(zeta(x+i*y)) with colormap
```

[3] When color maps are plotted on three-dimensional graphs, they appear in a flat plane on one of the back faces of the plot selected using the `axes` modifier to the `plot` command, and the c_1-axis associated with each are entirely independent of the plot's z-axis.



The `set c1range` command sets the range of values of c_1 to be assigned colors between black and white. By default, the lowest and highest values of c_1 found in the color map is assigned to black and white.

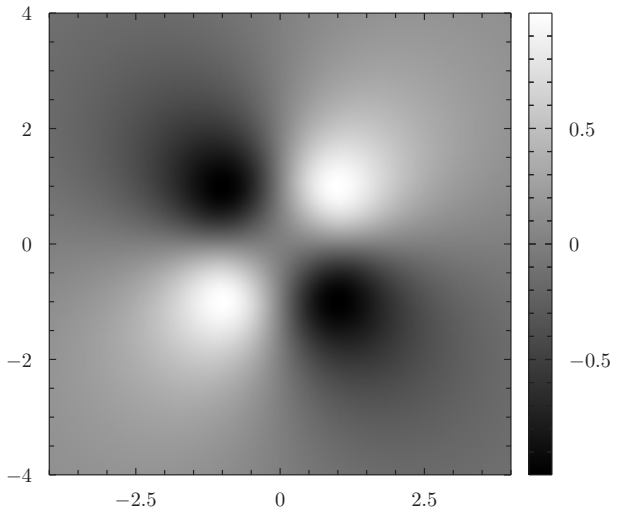
The `set c1format` command controls the format of the axis labels placed along the color scale bar on the right-hand side of the plot. In this case, they are marked as multiples of π.

The `set samples grid` command sets the dimensions of the grid of samples – or pixels – used to render the color map. If either value is replaced with an asterisk (`*`) then the current number of samples set in the `set samples` command is substituted.

If a data file is supplied to the `colormap` plot style, then the datapoints need not lie on the specified regular grid, but are first re-sampled onto this grid using the interpolation method specified using the `set samples interpolate` command (see Section 5.7). Three methods are available. `nearestNeighbor` uses the value of c_1 associated with the datapoint closest to each grid point, producing color maps which look like Voronoi diagrams. `inverseSquare` interpolation returns a weighted average of the supplied data points, using the inverse squares of their distances from each grid point as weights. `monaghanLattanzio` interpolation uses the weighting function of Monaghan & Lattanzio (1985) which is described further in Section 5.7).

In the following example, a color map of a quadrupole is produced using four input datapoints:

```
set nokey
set size 8 square
set samples grid 200x200 interpolate inverseSquare
plot [-4:4][-4:4] '--' with colormap
-1 -1  1
-1  1 -1
1 -1 -1
1  1  1
END
```



8.12.1 Custom color mappings

The default mapping used between values of c_1 and color is a grayscale mapping. This is scaled such that the smallest value of c_1 in the map corresponds to black, and largest value corresponds to white.

Alternatively, the user can supply any algebraic expressions for converting values of c_1 into colors. Moreover, these custom color mappings need not be one-parameter mappings depending only on a single variable c_1, but can depend on up to four quantities c_1, c_2, c_3 and c_4. This makes it possible, for example, to represent both the amplitude and complex phase of a quantity in a single color map.

Pyxplot's `colormap` plot style takes between three and seven columns of data, which may be supplied either from one or more function(s), or from a data file. If data is read from a data file, then the first two columns of output data are assumed to contain the respective positions of each datapoint along the x-axis and the y-axis. The next column contains the value c_1, and may be followed by up to three further optional values c_2, c_3 and c_4.

In the case where one or more function(s) are supplied, they are assumed to be functions of both x and y, and are sampled at a grid of points in the (x, y) plane; the first supplied function returns the value c_1, and may be followed by up to three further optional functions for c_2, c_3 and c_4..

The color mapping is set using the `set colormap` command, which takes an algebraic expression which should be a function of the variables `c1`, `c2`, `c3` and `c4`. This should evaluate either to a color object or a number (in which case a color is drawn from the current palette).

```
set colormap <expr> [ mask <expr> ]
```

If the optional mask expression is supplied, then any areas in a color map where this expression evaluates to false (e.g. zero) are made transparent. The following color mapping, which is the default, produces a grayscale color mapping of the third column of data supplied to the `colormap` plot style; further columns of data, if supplied, are not used:

```
set c1range [*:*] renormalise noreverse
set colormap gray(c1)
```

The `set c<n>range` command command specifies how the values of c_n are processed before being used in the expressions supplied to the `set colormap` command. It has the following syntax:

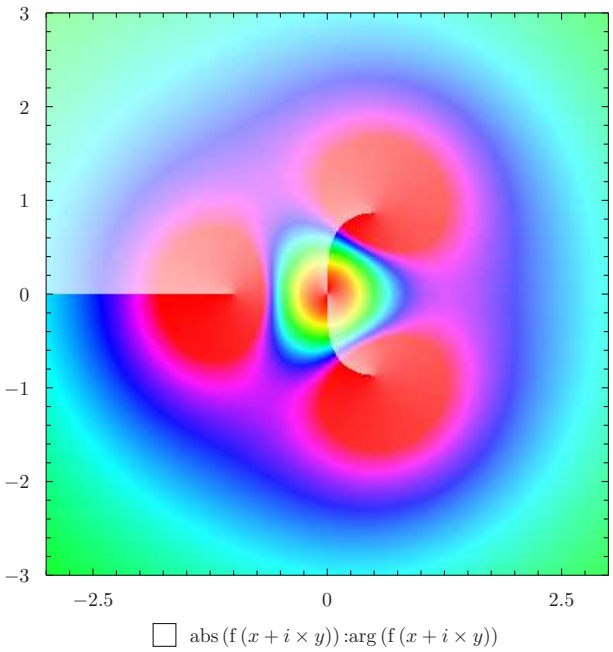
```
set c<n>range [ <range> ]
              [ reversed | noreversed ]
              [ renormalise | norenormalise ]
```

If the `renormalise` option is specified, then the values of c_n at each point in the data grid are first scaled into the range zero to one. This is often useful, since the color components passed to the `rgb()`, `gray()`, `hsb()` and `cmyk()` functions must be in this range. Thus, in the example given above, the lowest value of c_1 corresponds to black (i.e. brightness 0), and the highest value corresponds to white (i.e. brightness 1). If an explicit range is specified to the `set c<n>range` command, then the upper limit of this range maps to the value one, and the lower limit maps to the value zero. An asterisk (`*`) means that the lowest or highest value found in the map is substituted. The mapping is inverted if the `reverse` option is specified, such that the upper limit maps to zero, and the lower limit maps to one. Intermediate values are scaled either linearly or logarithmically, and these behaviours can be selected with the following commands:

```
set logscale c1
set linearscale c1
```

In the example below, a color map of the function $f(z) = 3x^2/(x^3 + 1)$ is made, using hue to represent the magnitude of $f(z)$ and saturation to represent the complex argument of $f(z)$:

```
set numerics complex
set sample grid 400x400
set nogrid
set size square
set key below
set c1range[0:2]
set colmap hsb(c1,c2*0.7+0.3,1)
f(x) = 3*x**2 / (x**3+1)
plot [-3:3][-3:3] abs(f(x+i*y)):arg(f(x+i*y)) with colormap
```



In the next example, three circular pools of red, green, and blue illumination are overlapped to show how colors mix together:

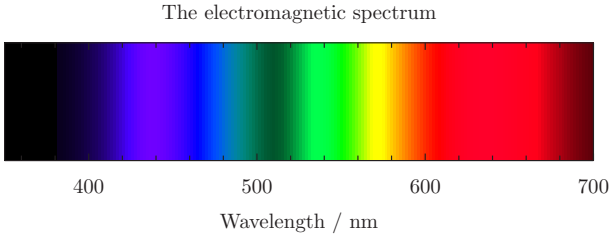
```
set size square
set nokey
set nocolkey
set numeric errors quiet
set noxtics ; set noytics
set axis x invisible
set axis y invisible
d    = 0.5
t(x) = max(0,2-exp(x**8))
set colmap rgb(t(hypot(c1  ,c2-d/sqrt(2))), \
t(hypot(c1+d,c2+d        )), \
t(hypot(c1-d,c2+d        ))  )
set sample grid 250x250
set c1range norenorm
set c2range norenorm
plot [-1.5:1.5][-1.5:1.5] x:y with colormap
```

The same is possible with CMYK colors, to demonstrate how substractive color mixing works:

```
set size square
set nokey
set nocolkey
set numeric errors quiet
set noxtics ; set noytics
set axis x invisible
set axis y invisible
d    = 0.5
t(x) = max(0,2-exp(x**8))
set colmap cmyk(t(hypot(c1  ,c2-d/sqrt(2))), \
t(hypot(c1+d,c2+d       )), \
t(hypot(c1-d,c2+d       )), \
0  )
set sample grid 250x250
set c1range norenorm
set c2range norenorm
plot [-1.5:1.5][-1.5:1.5] x:y with colormap
```

The function `colors.wavelength(wlen,normalisation)` provides a color representation of any given wavelength of light, useful for producing representations of the electromagnetic spectrum:

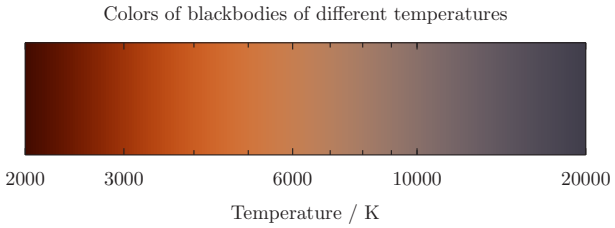
```
set nokey
set nocolkey
set size ratio 0.2
set noytics
set xlabel 'Wavelength'
set noylabel
set linear x y
set colmap colors.wavelength(c1,1)
set sample grid 200x2
set c1range norenorm
set title 'The electromagnetic spectrum'
plot [unit(350*nm):unit(700*nm)][0:1] x with colormap
```



The function `colors.spectrum(spectrum,normalisation)` takes a function as its first input, which should return a spectral energy distribution (in arbitrary units) as a function of wavelength. In this example, we show the colors of blackbodies of different temperatures. We renormalise their brightnesses by T^{-4} to avoid saturating hot blackbodies to white:

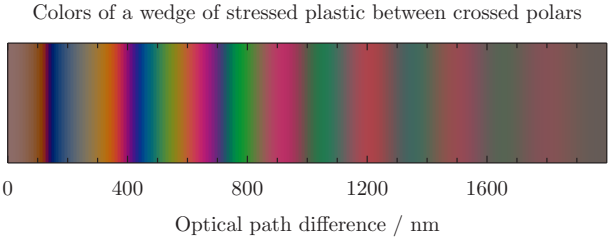
```
set log x
```

```
set linear y
set nokey
set nocolkey
set size ratio 0.2
set noytics
set xlabel 'Temperature'
set noylabel
f(lambda) = phy.Bv(phy.c/lambda,c1) / (c1 / unit(6000*K))**4
set colmap colors.spectrum(f,3e3)
set sample grid 200x2
set c1range norenorm
set title 'Colors of blackbodies of different temperatures'
plot [unit(2000*K):unit(20000*K)] x with colormap
```



As a final example, we use this function to plot the interference pattern seen when a wedge of stressed plastic, a birefrigent material, is viewed between crossed polars:

```
set nokey
set nocolkey
set size ratio 0.2
set noytics
set noylabel
spec(wl) = cos(2*pi*c1 / wl) * 1.25
set unit of length nm
set xlabel 'Optical path difference'
set colmap colors.spectrum(spec,1)
set sample grid 400x2
set c1range norenorm
set title 'Colors of a wedge of stressed plastic between crossed polars'
plot [unit(0*m):unit(2e-6*m)][0:1] x with colormap
```



8.12.2 Color scale bars

By default, plots with color maps with single-parameter color mappings are accompanied by color scale bars, which appear by default on the right-hand

side of the plot. Such scale bars may be configured using the `set colorkey` command. Issuing the command

```
set colorkey
```

by itself causes such a scale to be drawn on graphs in the default position, usually along the right-hand edge of the graphs. The converse action is achieved by:

```
set nocolorkey
```

The command

```
unset colorkey
```

causes Pyxplot to revert to its default behaviour, as specified in a configuration file, if present. A position for the key may optionally be specified after the `set colorkey` command, as in the example:

```
set colorkey bottom
```

Recognised positions are `top`, `bottom`, `left` and `right`. `above` is an alias for `top`; `below` is an alias for `bottom` and `outside` is an alias for `right`.

The format of the ticks along such scale bars may be set using the `set c1format` command command, which is similar in syntax to the `set xformat` command (see Section 8.8.8), but which uses `c` as its dummy variable.

The positions of the ticks along color scale bars may be set using the `set c1tics` command command, which has similar syntax to the `set xtics` command.

Example 25: An image of the Mandelbrot set.

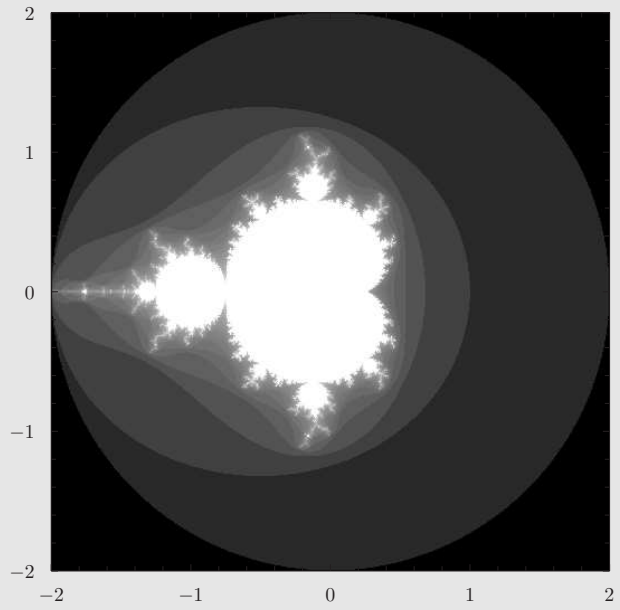
The Mandelbrot set is a set of points in the complex plane whose boundary forms a fractal with a Hausdorff dimension of two. A point c in the complex plane is defined to lie within the Mandelbrot set if the complex sequence of numbers

$$z_{n+1} = z_n^2 + c,$$

subject to the starting condition $z_0 = 0$, remains bounded.

The map of this set of points has become a widely-used image of the power of chaos theory to produce complicated structure out of simple algorithms. To produce a more pleasing image, points in the complex plane are often colored differently, depending on how many iterations n of the above series are required for $|z_n|$ to exceed 2. This is the point of no return, beyond which it can be shown that $|z_{n+1}| > |z_n|$ and that divergence is guaranteed. In numerical implementations of the above iteration, in the absence of any better way to prove that the iteration remains bounded for a certain value of c, some maximum number of iterations m is chosen, and the series is deemed to have remained bounded if $|z_m| < 2$. This is implemented in Pyxplot by the built-in mathematical function `fractal_mandelbrot(z,m)`, which returns an integer in the range $0 \leq i \leq m$.

```
set numerics complex
set sample grid 500x500
set size square
set nokey
set nocolkey
set log c1
plot [-2:2][-2:2] fractals.mandelbrot(x+i*y,70)+1 with colormap
```



8.13 Contour maps

Contour maps are similar to color maps, but instead of coloring the whole (x, y) plane, lines are drawn to indicate paths of constant $c(x, y)$. The number of contours drawn, and the values c_1 that they correspond to, is set using the `set contour` command, which has the following syntax:

```
set contours [ ( <number> |
               \( { <value> } \) ) ]
             [ (label | nolabel) ]
```

If `<number>` is specified, as in the example

```
set contours 8
```

then the specified number of contours are drawn at evenly spaced intervals. Whether the contours are linearly or logarithmically spaced can be changed using the commands

```
set logscale c1
set linearscale c1
```

By default, the range of values spanned by the contours is automatically scales to the range of the data provided. However, it may also be set manually using the `set c1range` command as in the example

```
set c1range [0:10]
```

The default autoscaling behaviour can be restored using the command

```
set c1range [*:*]
```

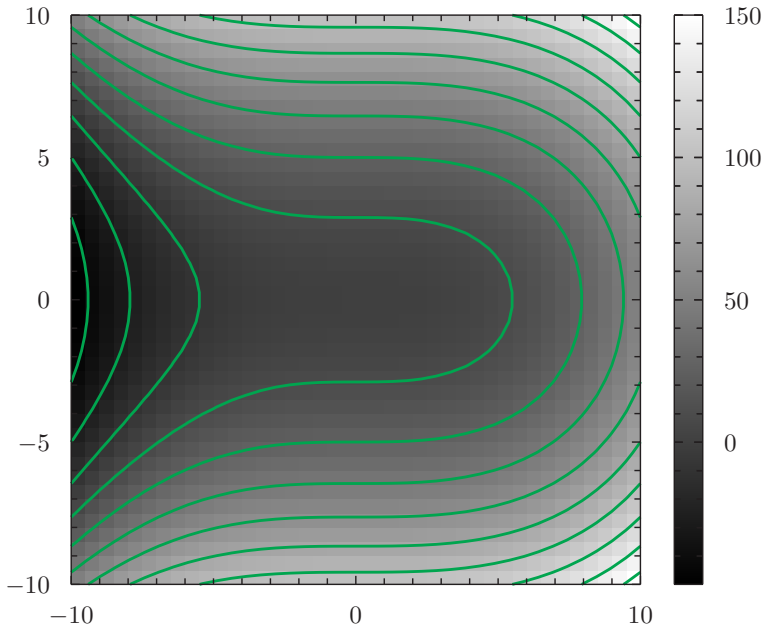
Alternatively, an explicit list of the values of c for which contours should be drawn may be specified to the `set contour` command as a ()-bracketed comma-separated list. For example:

```
set contours (0,5,10,20,40)
```

If the option `label` is specified to the `set contour` command, then each contour is labelled with the value of c that it corresponds to. If the option `nolabel` is specified, then the contours are left unlabelled.

In the following example, a contour map is overlaid on top of a color map of the function $x^3/20 + y^2$:

```
set nokey
set size 8 square
plot [-10:10][-10:10] x**3/20+y**2 with colormap, \
x**3/20+y**2 with contours col green lw 2 lt 1
```



The `contourmap` plot style differs from other plot styles in that it is not permitted to take expressions such as `$2+1` for style modifiers such as `linetype` (see Section 8.1) which use additional columns of input data to plot different points

in different styles. However, the variable `c1` may be used in such expressions to define different styles for different contours:

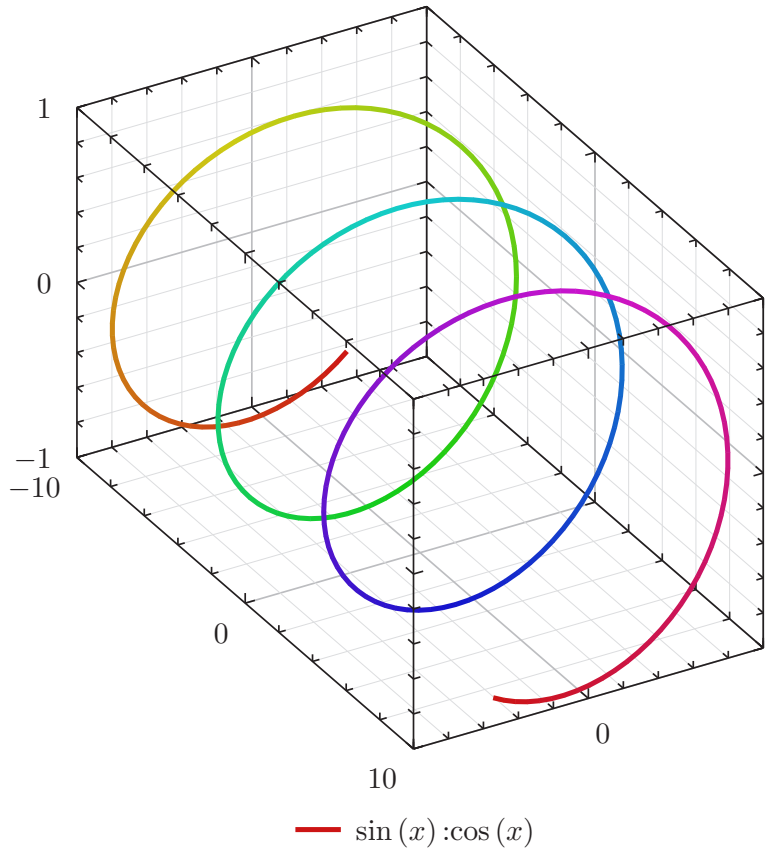
✗
```
plot 'datafile' with contourmap linetype $5
```

✔
```
plot 'datafile' with contourmap linetype c1/10
```

8.14 Three-dimensional plotting

Three-dimensional graphs may be produced by placing the modifier `3d` immediately after the `plot` command, as demonstrated by the following simple example which draws a helix:

```
set key below
set size 8 ratio 0.6 zratio 0.6
set grid
plot 3d sin(x):cos(x) with lw 3 col hsb(x/20+0.5,0.9,0.8)
```



Many plot styles take additional columns of data when used on three-dimensional plots, reading in three values for the x, y and z coordinates of each datapoint, where previously only x and y coordinates were required. In the above

example, the `lines` plot style is used, which takes three columns of input data when used on three-dimensional plots, as compared to two on two-dimensional plots. The descriptions of each plot style in Section 8.2 includes information on the number of columns of data required for two- and three-dimensional plots.

The example above also demonstrates that the `set size` command takes an additional aspect ratio `zratio` which affects three-dimensional plots; whereas the aspect ratio `ratio` determines the ratio of the lengths of the y-axes of plots to their x-axes, the aspect ratio `zratio` determines the ratio of the lengths of the z-axes of plots to their x-axes.

The angle from which three-dimensional plots are viewed can be set using the `set view` command. This should be followed by two angles, which can either be expressed in degrees, as dimensionless numbers, or as quantities with physical units of angle:

```
set view 60,30

set unit angle nodimensionless
set view unit(0.1*rev),unit(2*rad)
```

The orientation $(0, 0)$ corresponds to having the x-axis horizontal, the z-axis vertical, and the y-axis directed into the page. The first angle supplied to the `set view` command rotates the plot in the (x, y) plane, and the second angle tips the plot up in the plane containing the z-axis and the normal to the user's two-dimensional display.

The `replot` command command may be used to add additional datasets to three-dimensional plots in an entirely analogous fashion to two-dimensional plots.

8.14.1 Surface plotting

The `surface` plot style is similar to the `colormap` and `contourmap` plot styles, but produces maps of the values $z(x, y)$ of functions of two variables using three-dimensional surfaces. The surface is displayed as a grid of four-sided elements, whose number may be specified using the `set samples` command, as in the example

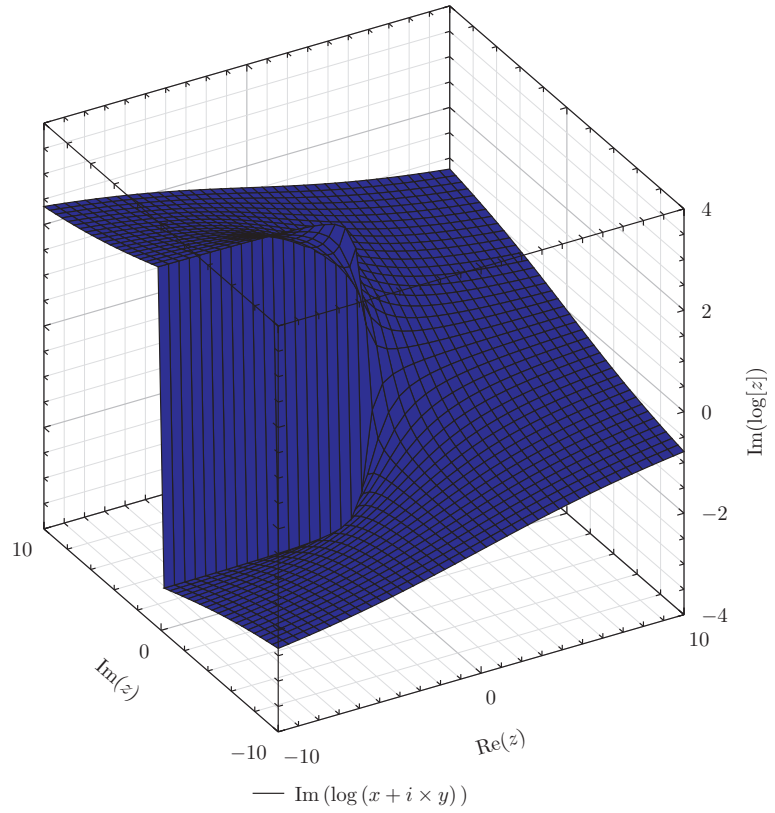
```
set samples grid 40x40
```

If data is supplied from a data file, then it is first re-sampled onto a regular grid using one of the methods described in Section 8.12.

The example below plots a surface indicating the magnitude of the imaginary part of $\log(x + iy)$:

```
set numerics complex
set xlabel r"Re($z$)"
set ylabel r"Im($z$)"
set zlabel r"$\mathrm{Im}(\mathrm{log}[z])$"
set key below
set size 8 square
set grid
set view -30,30
```

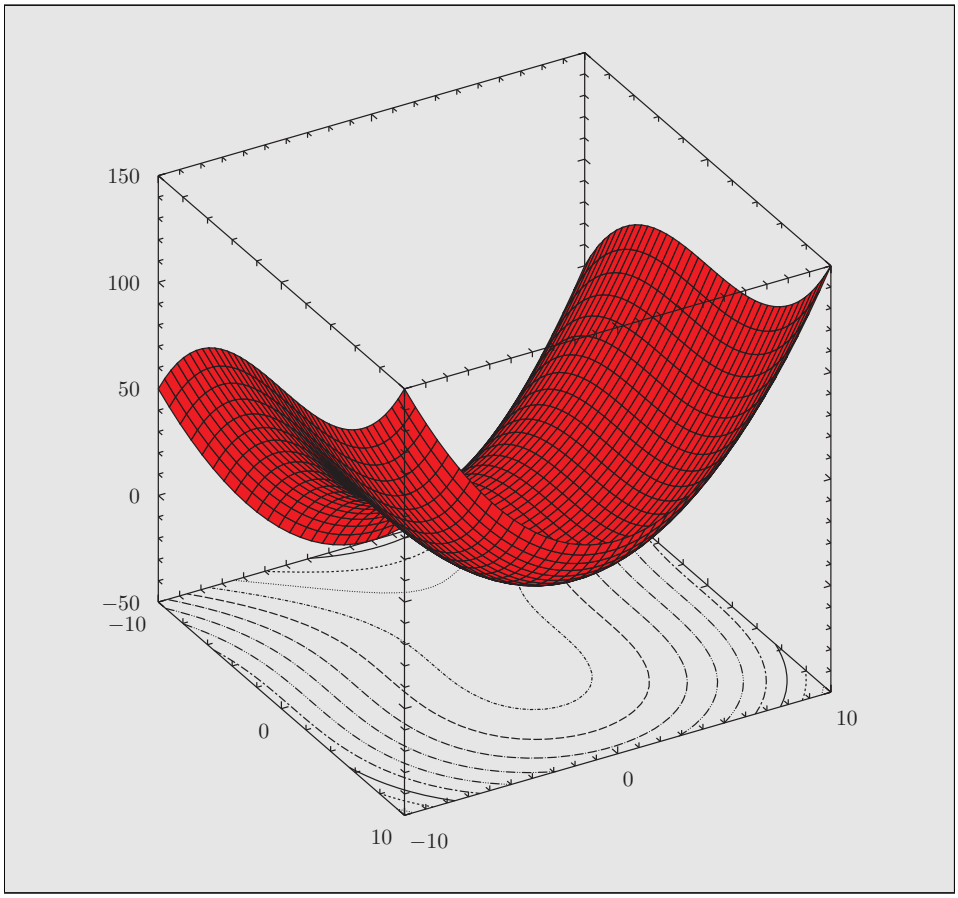
```
plot 3d [-10:10][-10:10] Im(log(x+i*y)) \
with surface col black fillcol blue
```



Example 26: A surface plotted above a contour map.

In this example, we plot a surface showing the value of the expression $x^3/20+y^2$, and project below it a series of contours in the (x, y) plane.

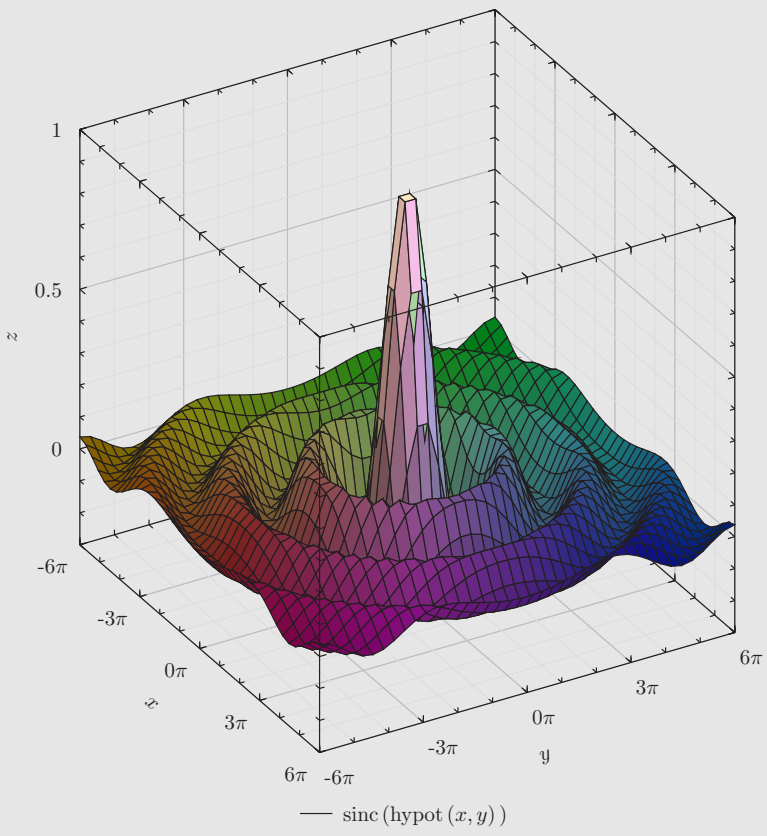
```
set nokey
set size 8 square
plot 3d x**3/20+y**2 with surface col black fillc red, \
x**3/20+y**2 with contours col black
```



Example 27: The sinc(x) function represented as a surface.

In this example, we produce a surface showing the function sinc(r) where $r = \sqrt{x^2 + y^2}$. To produce a prettier result, we vary the color of the surface such that the hue of the surface varies with azimuthal position, its saturation varies with radius r, and its brightness varies with height z.

```
set numerics complex
set xlabel "$x$"
set ylabel "$y$"
set zlabel "$z$"
set xformat r"%s$\pi$"%(x/pi)
set yformat r"%s$\pi$"%(y/pi)
set xtics 3*pi ; set mxtics pi
set ytics 3*pi ; set mytics pi
set ztics
set key below
set size 8 square
set grid
plot 3d [-6*pi:6*pi][-6*pi:6*pi][-0.3:1] sinc(hypot(x,y)) \
with surface col black \
fillcol hsb(atan2($1,$2)/(2*pi)+0.5,hypot($1,$2)/30+0.2,$3*0.5+0.5)
```



Chapter 9

Producing image files

Pyxplot is able to produce graphical output in a wide range of image formats, including both vector graphic formats such as PostScript and scalable vector graphics (`svg`), and rasterised formats such as bitmap (`bmp`) and jpeg. Additionally, it can produce graphical output for immediate preview on screen. In this chapter we describe how to select and control which image format should be used.

9.1 The `set terminal` command

The `set terminal` command is used to select the image format in which output should be produced, and also to specify a range of fine controls such as whether output should be in color or black-and-white. In its simplest usage, the command is followed by the name of the output image format which is to be used, which may be any of the options listed in Table 9.1.

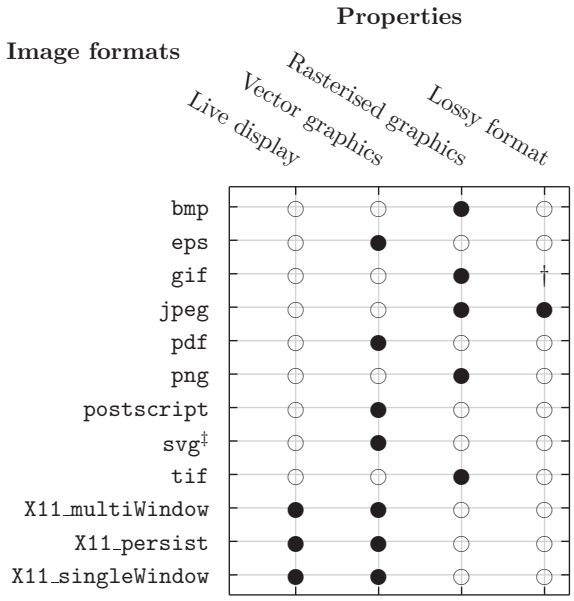
9.1.1 Previewing graphs on the screen

Three output terminal produce immediate previews to the screen: `X11_singleWindow`, `X11_persist`, `X11_multiWindow`. The default of these options – i.e. the default terminal when Pyxplot is started up in interactive mode – is `X11_singleWindow`. In this terminal, each time a new plot is generated, if the previous plot is still open on the display, the old plot is replaced with the new one. This way, only one plot window is open at any one time. This behaviour is intended to prevent the desktop from becoming flooded with plot windows.

The alternative `X11_multiWindow` terminal is similar in all respects except that each new plot is generated in a new window, regardless of whether any previous plots are still open on the display. This is especially useful when multiple plots are to be compared side-by-side:

```
set terminal X11_singleWindow
plot 'data1.dat'
plot 'data2.dat'  <-- first plot window disappears
```

c.f.:



† – Although the `gif` image format is not lossy in the traditional sense, it reduces the number of colors to a palette of no more than 256 unique colors. Thus, whilst it is lossless for images which only contain small numbers of unique colors, some color distortion may occur in colorful images.

‡ – The `svg` terminal is experimental and may be unstable. It relies upon the use of the `svg` output device in ghostscript, which may not be present on all systems.

Table 9.1: A list of the properties of the graphical output formats supported by Pyxplot.

```
set terminal X11_multiWindow
plot 'data1.dat'
plot 'data2.dat'  <-- first plot window remains
```

The third of these terminals, `X11_persist`, is similar to `X11_multiWindow` but keeps plot windows open after Pyxplot terminates in distinction from the above two terminals, which close all plot windows upon exit.

9.1.2 Producing images on disk

The remaining terminals listed in Table 9.1 direct graphical output to disk in a selection of rasterised and vector graphics formats. The filename of the resulting image file may be set using the `set output` command, as in the example:

```
set output 'my_plot.eps'
```

Use of rasterised image formats inevitably results in some loss of image quality since the plot has to be rasterised into a bitmapped graphic image. By default, this rasterisation is performed at a resolution of 300 dpi, though this may be changed using the `set terminal dpi` command, which should be followed by a numerical value. Alternatively, the resolution may be changed using the `DPI` option in the `settings` section of a configuration file (see Chapter 19).

9.1.3 The complete syntax of the `set terminal` command

In addition to being used to select the graphical format in which output should be produced, the `set terminal` command takes many options for fine-tuning the behaviours of particular terminals. Its complete syntax is:

```
set terminal ( X11_singleWindow | X11_multiWindow | X11_persist |
               bmp | eps | gif | jpeg | pdf | png | postscript |
               svg | tiff )
             ( color | colour | monochrome )
             ( dpi <value> )
             ( portrait | landscape )
             ( invert | noinvert )
             ( transparent | solid )
             ( antialias | noantialias )
             ( enlarge | noenlarge )
```

The following table lists the effects which each of these settings has:

`X11_singleWindow`	Displays plots on the screen (in X11 windows, using ghostviewor other viewing application selected using the `set viewer` command). Each time a new plot is generated, it replaces the old one, to prevent the desktop from becoming flooded with old plots.[1] **[default when running interactively; see below]**

[1] The authors are 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 Pyxplot closes.
`bmp`	Sends output to a Windows bitmap (`.bmp`) file. The filename for this file should be set using `set output`. This is a bitmap graphics terminal.
`eps`	As above, but produces Encapsulated PostScript.
`gif`	As above, but produces a gif image. This is a bitmap graphics terminal.
`jpeg`	As above, but produces a jpeg image. This is a bitmap graphics terminal.
`pdf`	As above, but produces pdf output.
`png`	As above, but produces a png image. This is a bitmap graphics terminal.
`postscript`	As above, but sends output to a PostScript file. [**default when running non-interactively; see below**]
`svg`	As above, but produces an svg image.[2]
`tiff`	As above, but produces a tiff image. This is a bitmap graphics terminal.
`color`	Allows datasets to be plotted in color. Automatically they will be displayed in a series of different colors, or alternatively colors may be specified using the `with color` plot modifier (see below). [**default**]
`color`	Equivalent US spelling of the above.
`monochrome`	Opposite to the above; all datasets will be plotted in black by default.
`dpi`	Sets the number of dots per inch at which rasterised graphic output should be sampled (i.e. the output image resolution)
`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 bitmap output terminals identified above – i.e. the `bmp`, `gif`, `jpeg`, `png` and `tiff` terminals – which produces output with inverted colors.[3]
`noinvert`	Modifier for the bitmap output terminals identified above; 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**]

[2] The `svg` output terminal is experimental and may be unstable. It relies upon the use of the `svg` output device in Ghostscript, which may not be present on all systems.

[3] 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 textcolor` and `set axescolor` commands (see Section 8.9).

`antialias`	Modifier for the bitmap output terminals identified above; produces antialiased output, with color boundaries smoothed to disguise the effects of pixelisation [**default**]
`noantialias`	Modifier for the bitmap output terminals identified above; opposite to the above
`enlarge`	Enlarge or shrink contents to fit the current paper size.
`noenlarge`	Do not enlarge output; opposite to the above. [**default**]

9.2 The default terminal

The default terminal is normally `X11_singleWindow`. There are two exceptions to this. When Pyxplot is not called from within an X11 session[4], and it therefore cannot open graphical display windows, the default terminal changes to `eps`. 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_persist` terminal becomes default, since it does not close plot windows when Pyxplot exits.

9.3 PostScript output

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.

9.3.1 Paper sizes

By default, the `postscript` terminal, and the `enlarge` terminal option, read the paper size for their output from the user's system locale settings. It may be changed, however, with `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 complete list of recognised paper size names can be found in Appendix 16.[5]

9.4 Backing up over-written files

By default, when graphical output is sent to a file – i.e. a PostScript file or a bitmap image – any pre-existing file is overwritten if its filename matches that of the file which Pyxplot generates. This behaviour may be changed with the `set backup` command, which has the syntax:

[4] i.e. the environment variable `DISPLAY` is not set.

[5] Marcus Kuhn has written a very complete treatise on international paper sizes, which can be downloaded from: http://www.cl.cam.ac.uk/~mgk25/iso-paper.html. Further details on the Swedish extensions to this system, and the Japanese B-series, can be found on Wikipedia: http://en.wikipedia.org/wiki/Paper_size.

```
set backup
set nobackup
```

When this switch is turned on, pre-existing files will be renamed with a tilde (∼) appended to their filenames, rather than being overwritten.

9.5 Changing font

The font used by Pyxplot can be changed using the `set preamble` command. In latex, fonts are changed by adding a header to a document, and the `set preamble` command specifies text that should be passed to latex before rendering any of the text labels on the present canvas. For example, the following command changes the default font to sans-serif:

```
set preamble r"\renewcommand{\familydefault}{\sfdefault}"
```

Chapter 10

Producing vector graphics

This chapter provides a systematic description of how Pyxplot can be used to produce general-purpose vector graphics. It also describes how to produce galleries of multiple graphs side-by-side, together with how graphs may be annotated with text or arrows. For more information about how to produce graphical output in various image formats, see Section 9.1. For more information about graph plotting, see Chapter 8.

10.1 Adding other vector graphics objects

In addition to graphs, a range of other objects can be placed on graphics canvas:

- Rectangular boxes (the `box` command).
- Circles and arcs (the `arc` and `circle` commands).
- Ellipses and elliptical arcs (the `ellipse` command).
- Graphical images in `bmp`, `eps`, `gif`, `jpeg` or `png` formats (the `eps` and `image` commands).
- Lines and arrows (the `arrow` and `line` commands).
- Piecharts (the `piechart` command).
- Points labelled by crosses and other symbols (the `point` command).
- Polygons (the `polygon` command).
- Text labels (the `text` command).

Put together, these commands can be used to produce a wide range of vector graphics. The remainder of this chapter describes each of these commands in turn, providing a variety of examples of their use.

These commands all interface with Pyxplot's maths environment in common ways. For example, vector positions on the graphics canvas may be specified in three ways:

- Two comma-separated dimensionless numbers, taken to be in centimetres, e.g. `4,5`.
- Two comma-separated numbers with dimensions of length, e.g. `45*unit(mm), 13*unit(cm)`.
- As a vector, either dimensionless, or with units of length, e.g. `vector(4,5)` or `vector(13,25)*unit(mm)`.

Many of these commands take rotation angles as inputs: these may be specified either as dimensionless numbers, taken to be in degrees, or as values with physical units of angle, e.g. `0.25*unit(rev)`.

Where these commands take colors as inputs, as elsewhere in Pyxplot, the color may be specified in one of three ways:

- As a number, referred to a color from a present palette; see Section 8.1.1.
- As a recognised named color, e.g. `red`; see Section 19.4 for a list of these.
- As an object of type color, for example, `rgb(0,1,0)`, `hsb(0.5,0.5,0.5)`, `gray(0.2)`, `colors.green + colors.red`, `colors.yellow - colors.green`.

10.2 Multiplot mode

Pyxplot has two modes in which it can produce graphical output. In *singleplot* mode, the default, each time the `plot` command is issued, the canvas is wiped clean and the new plot is placed alone on a blank page. In *multiplot* mode, vector graphics objects accumulate on the canvas. Each time the `plot` command is issued, the new graph is placed on top of any other objects which were already on the canvas, and many plots can be placed side-by-side.

The user can switch between these two modes of operation by issuing the commands `set multiplot` and `set nomultiplot`. The `set origin` command is required for multiplot mode to be useful when placing plots side-by-side: it sets the position on the page of the lower-left corner of the next plot. It takes a comma-separated (x, y) coordinate pair, which may have units of length, or, if dimensionless, are assumed to be measured in centimetres. The following example plots a graph of $\sin(x)$ to the left of a plot of $\cos(x)$:

```
set multiplot
set width 8
plot sin(x)
set origin 10,0
plot cos(x)
```

10.3 The `text` command

Text labels may be added to multiplot canvases 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 color may optionally be specified with the `with color` modifier, as well as a rotation angle 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 with color red
```

The commands `set textcolor`, `set texthalign` and `set textvalign` can be used to set the color and alignment of the text produced with the `text` command. Alternatively, the `text` command takes three modifiers to control the alignment of the text which override these `set` commands. The `halign` and `valign` modifiers may be followed by any of the settings which may follow the `set texthalign` and `set textvalign` commands respectively, as in the following examples:

```
text 'This is some text' at 0,0 halign left valign top
text 'This is some text' at 0,0 halign right valign centre
```

The `gap` modifier allows a gap to be inserted in the alignment of the text. For example, the string `halign left gap 3*unit(mm)` would cause text to be rendered with its left side 3 mm to the right of the position specified for the text. This is useful for labelling points on diagrams, where the labels should be slightly offset from the points that they are associated with. If the `gap` modifier is followed by a dimensionless number, rather than one with dimensions of lengths, then it is assumed to be measured in centimetres.

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 which can subsequently be imported into documents, slideshows or webpages.

10.4 The `arrow` and `line` commands

Arrows may also be added to multiplot canvases 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 line type, line width and color of the arrow, may be specified using the same syntax as used in the plot command, using the `linetype`, `linewidth` and `color` modifiers after the word `with`, as in the example:

```
arrow from 0,0 to 10,10 \
with linetype 2 linewidth 5 color red
```

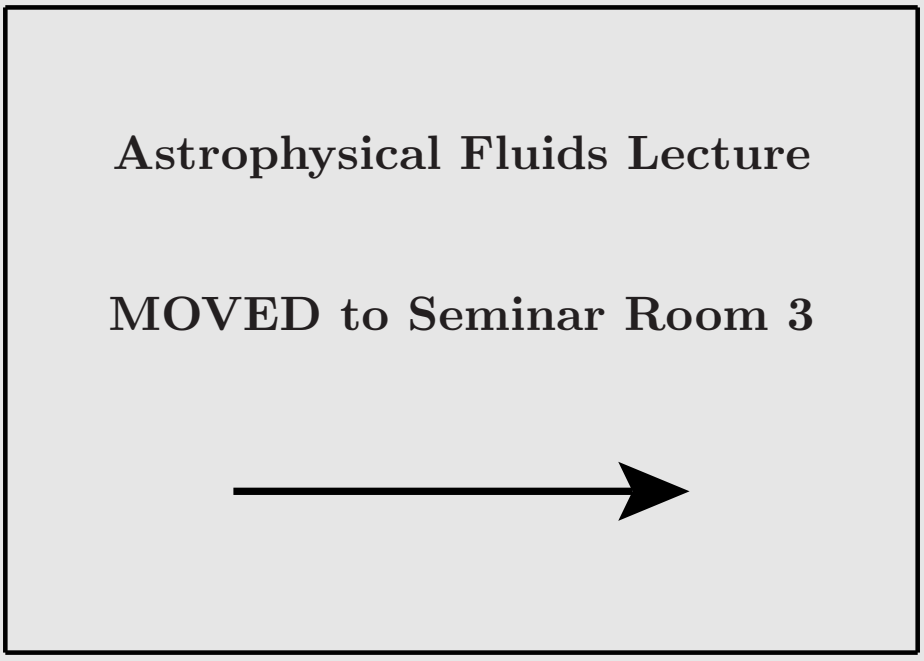
The style of the arrow may also be specified after the word `with`, and three options are available: `head` (the default), `nohead`, which produces line segments with no arrowheads on them, and `twoway`, which produces bidirectional arrows with heads on both ends.

The `arrow` command has a twin, the `line` command, which has the same syntax but with a different style setting of `nohead`.

Example 28: A simple notice generated with the `text` and `arrow` commands.

In this example script, we use Pyxplot's `arrow` and `text` commands to produce a simple notice advertising that a lecture has moved to a different seminar room:

```
set multiplot ; set nodisplay

w = unit(20*cm) # Width of notice
h = w/sqrt(2) # Height of notice

# Put a rectangular box around notice
line from 0,0 to w,0 with linewidth 5
line from w,0 to w,h with linewidth 5
line from w,h to 0,h with linewidth 5
line from 0,h to 0,0 with linewidth 5

# Write text of notice
set texthalign center ; set fontsize 3
text r"\bf Astrophysical Fluids Lecture" at w/2,3/4*h
text r"\bf MOVED to Seminar Room 3" at w/2, h/2
arrow from w/4, h/4 to 3/4*w, h/4 with linewidth 8

# Display notice
set display ; refresh
```



10.5 Editing items on the canvas

All objects on a multiplot canvas have a unique identification number. By default, these count up from one, such that the first item placed on the canvas is number one, the next is number two, and so forth. Alternatively, the user may specify a particular number for a particular object by supplying the modifier `item` to the `plot` command, followed by an integer identification number, as in the following example:

```
plot item 6 'data.dat'
```

If there were already an object on the canvas with identification number 6, this object would be deleted and replaced with the new object.

A list of all of the objects on the current multiplot canvas can be obtained using the `list` command, which produces output in the following format:

```
# ID   Command
    1  plot item 1 'data1.dat'
    2  plot item 2 'data2.dat'
    3  [deleted] plot item 3 'data3.dat'
```

A multiplot canvas can be wiped clean by issuing the `clear` command, which removes all items currently on the canvas. Alternatively, individual items may be removed using the `delete` command, which should be followed by a comma-separated list of the identification numbers of the objects to be deleted. Deleted items may be restored using the `undelete` command, which likewise takes a comma-separated list of the identification numbers of the objects to be restored, e.g.:

```
delete 1,2
undelete 2
```

Once a canvas has been cleared using the `clear` command, however, there is no way to restore it. Objects may be moved around on the canvas using the `move` command. For example, the following would move item 23 to position $(8, 8)$ measured in inches:

```
move 23 to 8*unit(in), 8*unit(in)
```

10.5.1 Settings associated with multiplot items

Of the settings which can be set with the `set` command, some refer to Pyxplot's global environment and whole multiplot canvases. Others, such as `set width` and `set origin` refer specifically to individual graphs and vector graphics items. For this reason, whenever a new multiplot graphics item is produced, it takes a copy of the settings which are specific to it, allowing these settings to be changed by the user before producing other multiplot items, without affecting previous items. The settings associated with a particular multiplot item can be queried by passing the modifier `item` to the `show` command, followed by the integer identification number of the item, as in the examples:

```
show item 3 width    # Shows the width of item 3
show item 3 settings # Shows all settings associated with item 3
```

The settings associated with a particular multiplot item can be changed by passing the same `item` modifier to the `set` command, as in the example, which sets the width of item 3 to be 10 cm:

```
set item 3 width 10*unit(cm)
```

After making such changes, the `refresh` command is useful: it produces a new graphical image of the current multiplot to reflect any settings which have been changed. The following example would produce a pair of plots, and then change the color of the text on the first plot:

```
set multiplot
plot f(x)
set origin 10,0
plot g(x)
set item 1 textcolor red
refresh
```

Another common use of the `refresh` command is to produce multiple copies of an image in different graphical formats. For example, having just developed a multiplot canvas interactively in the `X11_singlewindow`, copies can be produced as `eps` and `jpeg` images using the following commands:

```
set terminal eps
set output 'figure.eps'
refresh
set terminal jpeg
set output 'figure.jpg'
refresh
```

10.5.2 Reordering multiplot items

Items on multiplot canvases are drawn in order of increasing identification number, and thus items with low identification numbers are drawn first, at the back of the multiplot, and items with higher identification numbers are later, towards the front of the multiplot. When new items are added, they are given higher identification numbers than previous items and appear at the front of the multiplot.

If this is not the desired ordering, then the `swap` command may be used to rearrange items. It takes the identification numbers of two multiplot items and swaps their identification numbers and hence their positions in the ordered sequence. Thus, if, for example, the corner of item 3 disappears behind the corner of item 5, when the converse effect is actually desired, the following command should be issued:

```
swap 3 5
```

10.5.3 The construction of large multiplots

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 a great speed increase can be achieved by wrapping the script with:

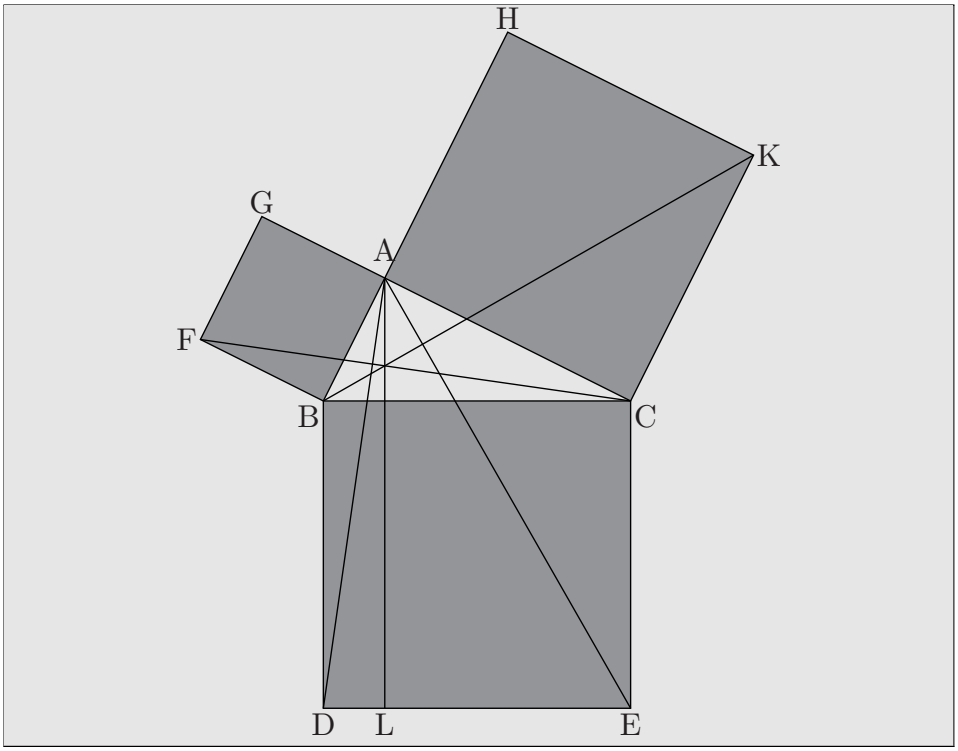
```
set nodisplay
[...prepare large multiplot...]
set display
refresh
```

Example 29: A diagram from Euclid's *Elements*.

In this more extended example script, we use Pyxplot's `arrow` and `text` commands to reproduce a diagram illustrating the 47th Proposition from Euclid's First Book of *Elements*, better known as Pythagoras' Theorem. A full text of the proof which accompanies this diagram can be found at http://www.gutenberg.org/etext/21076.

```
set unit angle nodimensionless
set multiplot ; set nodisplay

# Lengths of three sides of triangle
AB = 2*unit(cm)
AC = 4*unit(cm)
BC = hypot(AC, AB) # Hypotenuse
CBA = atan2(AC, AB) # Angle CBA

# Positions of three corners of triangle
Bx = 0*unit(cm)       ; By = 0*unit(cm) # The origin
Cx = Bx + BC          ; Cy = By
Ax = Bx + AB*cos(CBA) ; Ay = By + AB*sin(CBA)

# Positions of constructed points
Dx = Bx               ; Dy = -BC
Lx = Ax               ; Ly = Dy
Ex = Cx               ; Ey = Dy

Hx = Bx + (AB + AC) * cos(CBA)
Hy = By + (AB + AC) * sin(CBA)
Kx = Cx + (     AC) * cos(CBA)
Ky = Cy + (     AC) * sin(CBA)
```

```
Fx = Bx + AB*cos(CBA+90*unit(deg))
Fy = By + AB*sin(CBA+90*unit(deg))
Gx = Ax + AB*cos(CBA+90*unit(deg))
Gy = Ay + AB*sin(CBA+90*unit(deg))

# Construct diagram
box from Dx,Dy to Cx,Cy with fillcol gray50
box at Ax,Ay width AC height AC rot CBA-90*unit(deg) with fillcol gray50
box at Bx,By width AB height AB rot CBA with fillcol gray50
line from Bx,By to Kx,Ky
line from Fx,Fy to Cx,Cy
line from Ax,Ay to Dx,Dy
line from Ax,Ay to Lx,Ly
line from Ax,Ay to Ex,Ey

# Label diagram
set fontsize 1.3
TG = 0.5*unit(mm) # Gap left between labels and figure
text "A" at Ax,Ay gap TG*5 hal c val b
text "B" at Bx,By gap TG   hal r val t
text "C" at Cx,Cy gap TG   hal l val t
text "D" at Dx,Dy gap TG   hal c val t
text "E" at Ex,Ey gap TG   hal c val t
text "F" at Fx,Fy gap TG   hal r val c
text "G" at Gx,Gy gap TG   hal c val b
text "H" at Hx,Hy gap TG   hal c val b
text "K" at Kx,Ky gap TG   hal l val c
text "L" at Lx,Ly gap TG   hal c val t

# Display diagram
set display ; refresh
```

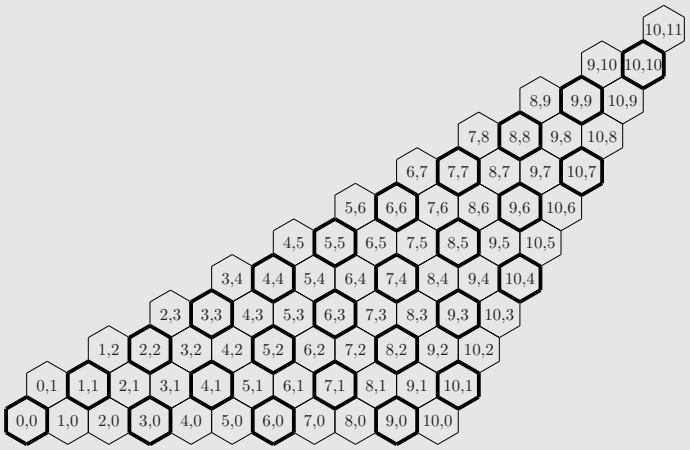


Example 30: A diagram of the conductivity of nanotubes.

In this example we produce a diagram of the *irreducible wedge* of possible carbon nanotube configurations, highlighting those configurations which are electrically conductive. We use Pyxplot's loop constructs to automate the production of the hexagonal grid which forms the basis of the diagram.

```
basisAngleX =   0*unit(deg)
basisAngleY = 120*unit(deg)
lineLen     =   5*unit(mm)

# Set up a transformation matrix
transformMat = matrix([[sin(basisAngleX),sin(basisAngleY)], \
[cos(basisAngleX),cos(basisAngleY)] ])
transformMat *= lineLen

subroutine line(p1,p2,lw)
{
line from transformMat*p1 to transformMat*p2 with linewid lw
}
```

```
subroutine hexagon(p,lw)
{
call line(p+vector([ 0, 0]),p+vector([ 0,-1]),lw)
call line(p+vector([ 0,-1]),p+vector([ 1,-1]),lw)
call line(p+vector([ 1,-1]),p+vector([ 2, 0]),lw)
call line(p+vector([ 2, 0]),p+vector([ 2, 1]),lw)
call line(p+vector([ 2, 1]),p+vector([ 1, 1]),lw)
call line(p+vector([ 1, 1]),p+vector([ 0, 0]),lw)
}

set multiplot ; set nodisplay

for x=0 to 10
{
for y=0 to x+1
{
p = vector([x+2*y , 2*x+y])
call hexagon(p, ((x-y)%3==0)?4:1)
text '%d,%d'%(x,y) at transformMat*(p+vector([1,0])) \
hal cen val cen
}
}

set display ; refresh
```



10.6 Linked axes and galleries of plots

In the previous chapter (Section 8.8.9), linked axes were introduced as a mechanism by which several axes on a single plot could be set to have the same range, or to be algebraically related to one another. Another common use for them is to make several plots on a multiplot canvas share common axes. Just as the following statement links two axes on a single plot to one another

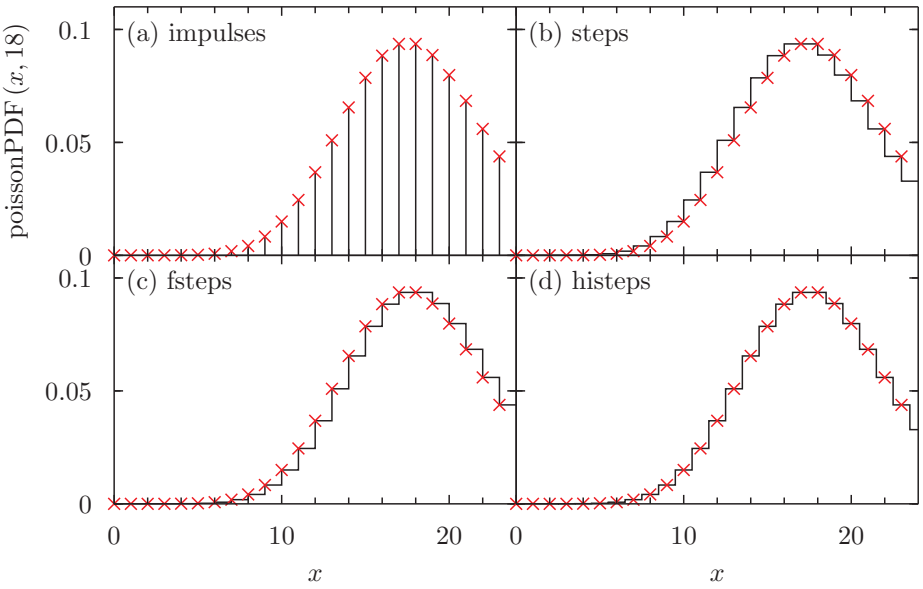
```
set axis x2 linked x
```

axes on the current plot can be linked to those of previous plots which are already on the multiplot canvas using syntax of the form:

```
set axis x2 linked item 2 x
```

A common reason for doing this is to produce galleries of side-by-side plots. The following series of commands would produce a 2×2 grid of plots, with axes only labelled along the bottom and left sides of the grid:

```
set multiplot
set nodisplay
width=5.4

set width width
set xrange [0:23.999]
set yrange [0:0.11]
set nokey
set texthalign left
set textvalign center

# Plot 1 (bottom left)
set xlabel "$x$"
set ylabel ""
set label 1 "(c) fsteps" at graph width*0.03 , graph width/goldenRatio*0.9
plot "barchart1.dat" with fsteps, "" with points

set axis x linked item 1 x
set axis y linked item 1 y

# Plot 2 (bottom right)
set origin 1*width, 0*width/goldenRatio
set yformat ""
set label 1 "(d) histeps" at graph width*0.03 , graph width/goldenRatio*0.9
plot "barchart1.dat" with histeps, "" with points

# Plot 3 (top left)
set origin 0*width, 1*width/goldenRatio
set xformat "" ; set xlabel ""
unset yformat ; set ylabel texify("poissonPDF(x,18)")
set label 1 "(a) impulses" at graph width*0.03 , graph width/goldenRatio*0.9
plot "barchart1.dat" with impulses, "" with points

# Plot 4 (top right)
set origin 1*width, 1*width/goldenRatio
set yformat "" ; set ylabel ""
set label 1 "(b) steps" at graph width*0.03 , graph width/goldenRatio*0.9
plot "barchart1.dat" with steps, "" with points

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



10.6.1 The `replot` command revisited

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 the function $g(x)$ to it:

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

Additionally, it is possible to modify any plot on the page by adding an `item` modifier to the `replot` statement to specify which plot should be replotted. The following example would produce two plots, and then add an additional function to the first plot:

```
set multiplot
plot f(x)
set origin 10,0
plot g(x)
replot item 1 h(x)
```

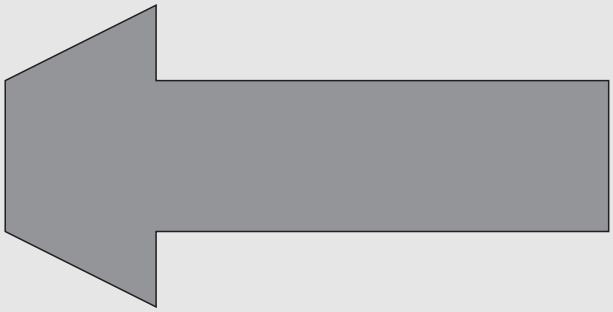
If no `item` number is specified, then the `replot` command acts by default upon the most recent plot to have been added to the multiplot canvas.

10.7 The `polygon` command

Example 31: A simple polygon.

In this simple example, we use Pyxplot's `polygon` command to generate a geometric shape from a list of points.

```
points = [ [0,-1], [0, 1], [2, 2], [2,1], [8,1], \
[8,-1], [2,-1], [2,-2]]

polygon points with fillcol gray50 col black
```

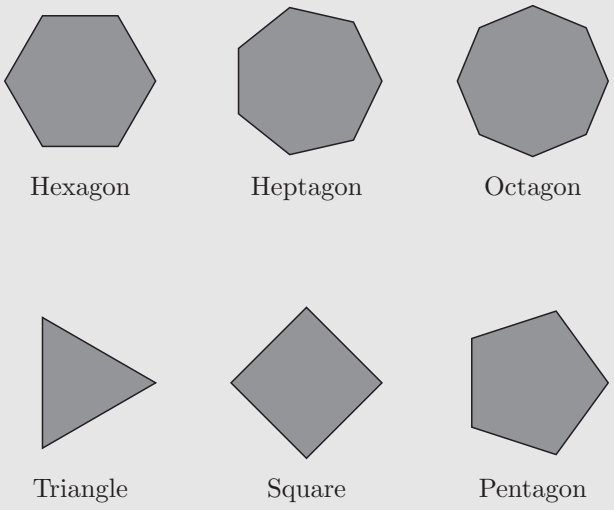


Example 32: The first eight regular polygons.

This example uses Pyxplot's flow control commands, together with its list methods and the `polygon` command, to generate a diagram of the first eight regular polygons.

```
rotate(a) = matrix( [cos(a), -sin(a)], \
[sin(a),  cos(a)] )

subroutine makePolygon(Nsides, centre)
{
points = []
for i=0 to Nsides
{
call points.append(centre + \
rotate(i/Nsides*unit(rev)) * vector(1,0))
}
polygon points with fillcol gray50 col black
}

set nodisplay ; set multiplot

set texthalign center
set textvalign top
```

```
foreach datum x,y,Nsides,name in "--"
{
call makePolygon(Nsides,vector(x,y))
text name at x,y-1.25
}
0 0 3 Triangle
3 0 4 Square
6 0 5 Pentagon
0 4 6 Hexagon
3 4 7 Heptagon
6 4 8 Octagon
END

set display ; refresh
```



10.8 The `image` command

Graphical images in `bmp`, `gif`, `jpeg` or `png` format may be placed on multiplot canvases using the `image` command[1]. In its simplest form, this has the syntax:

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

As an alternative to the `width` keyword the height of the image can be specified, using the analogous `height` keyword. An optional angle can also be specified using the `rotate` keyword; this causes the included image to be rotated counter-clockwise by a specified angle, measured in degrees. The keyword `smooth` may optionally be supplied to cause the pixels of the image to be interpolated[2].

[1] To maintain compatibility with historic versions of Pyxplot, the `image` command may also be spelt `jpeg`, with the identical syntax thereafter.

[2] Many commonly-used PostScript display engines, including Ghostscript, do not support this functionality.

Images which include transparency are supported. The optional keyword `notransparent` may be supplied to the `image` command to cause transparent regions to be filled with the image's default background color. Alternatively, an RGB color may be specified in the form `rgb<r>:<g>:<b>` after the keyword `transparent` to cause that particular color to become transparent; the three components of the RGB color should be in the range 0 to 255.

10.9 The eps command

Vector graphic images in eps format may be placed on multiplot canvases using the `eps` command, which has a syntax analogous to the `image` 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.

10.10 The box and circle commands

Rectangular boxes and circles may be placed on multiplot canvases using the `box` and `circle` commands, as in:

```
box from 0*unit(mm),0*unit(mm) to 25*unit(mm),70*unit(mm)
circle at 0*unit(mm),0*unit(mm) radius 70*unit(mm)
```

In the former case, two corners of the rectangle are specified, meanwhile in the latter case the centre of the circle and its radius are specified. The `box` command may also be invoked by the synonym `rectangle`. Boxes may be rotated using an optional `rotate` modifier, which may be followed by a counter-clockwise rotational angle which may either have dimensions of angle, or is assumed to be in degrees if dimensionless. The rotation is performed about the centre of the rectangle:

```
box from 0,0 to 10,3 rotate 45
```

The positions and dimensions of boxes may also be specified by giving the position of one of the corners of the box, together with its width and height. The specified corner is assumed to be the bottom-left corner if both the specified width and height are positive; other corners may be specified if the supplied width and/or height are negative. If such boxes are rotated, the rotation is about the specified corner:

```
box at 0,0 width 10 height 3 rotate 45
```

The line type, line width, and color of line with which the outlines of boxes and circles are drawn may be specified as in the `arrow` command, for example:

```
circle at 0,0 radius 5 with linetype 1 linewidth 2 color red
```

The shapes may be filled by specifying a `fillcolor`:

```
circle at 0,0 radius 5 with lw 10 color red fillcolor yellow
```

Example 33: A simple no-entry sign.

In this example script, we use Pyxplot's `box` and `circle` commands to produce a no-entry sign warning passers by that code monkeys can turn nasty when interrupted from their work.

```
set multiplot ; set nodisplay

w = 10 # Width of sign / cm

# Make no-entry sign
circle at 0,0 radius w with col null fillcol red
box from -(0.8*w),-(0.2*w) to (0.8*w),(0.2*w) \
with col null fillcol white

# Put a warning beneath the sign
set fontsize 3
set texthalign center ; set textvalign center
text r"\bf Keep Out! Code Monkey at work!" at 0,-1.2*w

# Display sign
set display ; refresh
```

Keep Out! Code Monkey at work!

10.11 The `arc` command

Partial arcs of circles may be drawn using the `arc` command. This has similar syntax to the `circle` command, but takes two additional angles, measured clockwise from the upward vertical direction, which specify the extent of the arc to be drawn. The arc is drawn clockwise from start to end, and hence the

following two instructions draw two complementary arcs which together form a complete circle:

```
set multiplot
arc at 0,0 radius 5 from -90 to   0 with lw 3 col red
arc at 0,0 radius 5 from   0 to -90 with lw 3 col green
```

If a `fillcolor` is specified, then a pie-wedge is drawn:

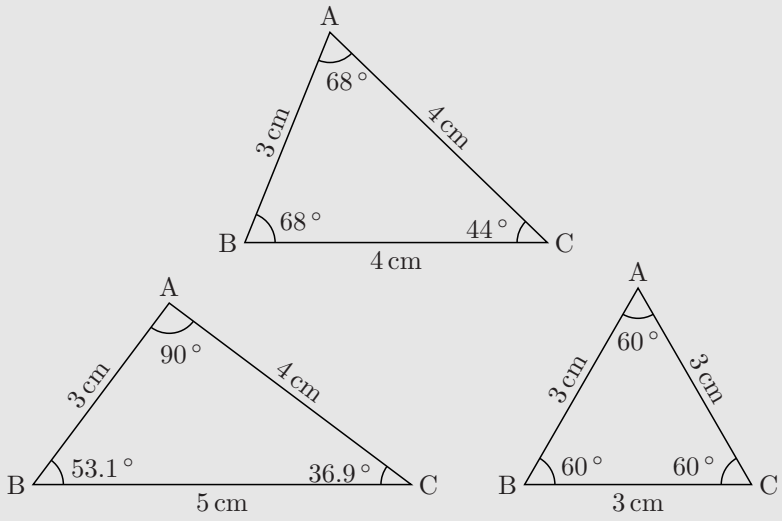
```
arc at 0,0 radius 5 from 0 to 30 with lw 3 fillcolor red
```

Example 34: Labelled diagrams of triangles.

In this example, we make a subroutine to draw labelled diagrams of the interior angles of triangles, taking as its inputs the lengths of the three sides of the triangle to be drawn and the position of its lower-left corner. The subroutine calculates the positions of the three vertices of the triangle and then labels them. We use Pyxplot's automatic handling of physical units to generate the latex strings required to label the side lengths in centimetres and the angles in degrees. We use Pyxplot's `arc` command to draw angle symbols in the three corners of a triangle.

```
set unit angle nodimensionless
set unit of length cm # Display lengths in cm
set unit of angle degree # Display angles in degrees
set numeric sigfig 3 display latex # Correct to 3 significant figure
cm  = unit(cm) # Shorthand to save space
deg = unit(deg)

turn(a) = matrix( [cos(a),-sin(a)], \
[sin(a), cos(a)]  )

# Define subroutine for drawing triangles
subroutine triangleDraw(B,AB,AC,BC)
{
# Use cosine rule to find interior angles
ABC = acos((AB**2 + BC**2 - AC**2) / (2*AB*BC))
BCA = acos((BC**2 + AC**2 - AB**2) / (2*BC*AC))
CAB = acos((AC**2 + AB**2 - BC**2) / (2*AC*AB))

  # Positions of three corners of triangle
C = B + vector(BC,0*cm)
A = B + turn(ABC)*vector(AB,0*cm)

  # Draw triangle
polygon [A,B,C]
```

```
  # Draw angle symbols
arcRad = 0.4*cm # Radius of angle arcs
arc at B radius arcRad from  90*deg-ABC to  90*deg
arc at C radius arcRad from -90*deg     to -90*deg+BCA
arc at A radius arcRad from  90*deg+BCA to 270*deg-ABC

  # Label lengths of sides
textGap = 0.1*cm
text "%s"%(BC) at (B+C)/2 gap textGap hal c val t
text "%s"%(AB) at (A+B)/2 gap textGap rot  ABC hal c val b
text "%s"%(AC) at (A+C)/2 gap textGap rot -BCA hal c val b

  # Label angles
arcRad2 = vector(1.4*arcRad , 0*cm)
text "%s"%CAB at A+turn(-90*deg+ABC-BCA)*arcRad2 hal c val t
text "%s"%ABC at B+turn(          ABC/2)*arcRad2 hal l val c
text "%s"%BCA at C+turn(180*deg - BCA/2)*arcRad2 hal r val c

  # Label points ABC
text "A" at A gap textGap hal c val b
text "B" at B gap textGap hal r val c
text "C" at C gap textGap hal l val c
}

# Display diagram with three triangles
set multiplot ; set nodisplay
call triangleDraw(vector([2.8,3.2])*cm, 3*cm, 4*cm, 4*cm)
call triangleDraw(vector([0.0,0.0])*cm, 3*cm, 4*cm, 5*cm)
call triangleDraw(vector([6.5,0.0])*cm, 3*cm, 3*cm, 3*cm)
set display ; refresh
```

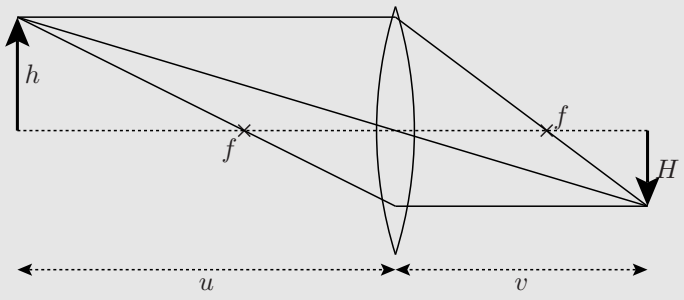


Example 35: A labelled diagram of a converging lens forming a real image.

In this example, we make a subroutine to draw labelled diagrams of converging lenses forming real images.

```
# Define subroutine for drawing lens diagrams
subroutine lensDraw(x0,y0,u,h,f)
{
# Use the thin-lens equation to find v and H
v = 1/(1/f - 1/u)
H = h * v / u

  # Draw lens
lc = 5.5*unit(cm) # Radius of curvature of lens
lt = 0.5*unit(cm) # Thickness of lens
la = acos((lc-lt/2)/lc) # Angular size of lens from center of curvature
lh = lc*sin(la) # Physical height of lens on paper
arc at x0-(lc-lt/2),y0 radius lc from  90*unit(deg)-la to  90*unit(deg)+la
arc at x0+(lc-lt/2),y0 radius lc from 270*unit(deg)-la to 270*unit(deg)+la
set texthalign right ; set textvalign top
point at x0-f,y0 label "$f$"
set texthalign left  ; set textvalign bottom
point at x0+f,y0 label "$f$"

  # Draw object and image
arrow from x0-u,y0 to x0-u,y0+h with lw 2
arrow from x0+v,y0 to x0+v,y0-H with lw 2
text "$h$" at x0-u,y0+h/2 hal l val c gap unit(mm)
text "$H$" at x0+v,y0-H/2 hal l val c gap unit(mm)

  # Draw construction lines
line from x0-u,y0 to x0+v,y0 with lt 2 # Optic axis
line from x0-u,y0+h to x0+v,y0-H # Undeflected ray through origin
line from x0-u,y0+h to x0,y0+h
line from x0,y0+h to x0+v,y0-H
line from x0+v,y0-H to x0,y0-H
line from x0,y0-H to x0-u,y0+h

  # Label distances u and v
ylabel = y0-lh-2*unit(mm)
arrow from x0-u,ylabel to x0,ylabel with twoway lt 2
arrow from x0+v,ylabel to x0,ylabel with twoway lt 2
text "$u$" at x0-u/2,ylabel hal c val t gap unit(mm)
text "$v$" at x0+v/2,ylabel hal c val t gap unit(mm)
}
```

```
# Display diagram of lens
set unit angle nodimensionless
set multiplot ; set nodisplay
call lensDraw(0*unit(cm),0*unit(cm), 5*unit(cm),1.5*unit(cm),2*unit(cm))
set display ; refresh
```



10.12 The `point` command

The `point` command places a single point on a multiplot canvases, in the same style which would be used when plotting a dataset on a graph with the `points` plotting style. It is useful for marking significant points on technical diagrams with crosses or other motifs.

The `point` command that the position of the point to be marked be specified after the `at` modifier. A text label to be attached next to the point may optionally be specified using the same `label` modifier as taken by the `plot` command. A `with` modifier may then be supplied, followed by any of the style modifiers: `color`, `pointlinewidth`, `pointsize`, `pointtype`, `style`.

The following example labels the origin as such:

```
set texthalign left
set textvalign centre
point at 0,0 label "The Origin" with ps 2
```

10.13 The `ellipse` command

Ellipses may be placed on multiplot canvases using the `ellipse` command. The shape of the ellipse may be specified in many different ways, by specifying

(i) the position of two corners of the smallest rectangle which can enclose the ellipse when its major axis is horizontal, together with an optional counter-clockwise rotation angle, applied about the centre of the ellipse. For example:

```
ellipse from 0,0 to 4,1 rot 70
```

(ii) the position of both the centre and one of the foci of the ellipse, together with any one of the following additional pieces of information: the ellipse's major axis length, its semi-major axis length, its minor axis length, its

semi-minor axis length, its eccentricity, its latus rectum, or its semi-latus rectum. For example:

```
ellipse focus 0,0 centre 2,2 majoraxis 4
ellipse focus 0,0 centre 2,2 minoraxis 4
ellipse focus 0,0 centre 2,2 ecc 0.5
ellipse focus 0,0 centre 2,2 LatusRectum 6
ellipse focus 0,0 centre 2,2 slr 3
```

(iii) the position of either the centre or one of the foci of the ellipse, together with any two of the following additional pieces of information: the ellipse's major axis length, its semi-major axis length, its minor axis length, its semi-minor axis length, its eccentricity, its latus rectum, or its semi-latus rectum. An optional counter-clockwise rotation angle may also be specified, applied about either the centre or one of the foci of the ellipse, whichever is specified. If no rotation angle is given, then the major axis of the ellipse is horizontal. For example:

```
ellipse centre 0,0 majoraxis 4 minoraxis 4
```

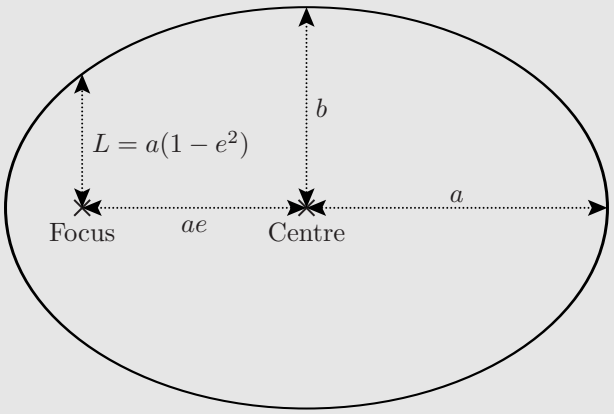
The line type, line width, and color of line with which the outlines of ellipses are drawn may be specified after the keyword `with`, as in the `box` and `circle` commands above. Likewise, ellipses may be filled in the same manner.

Example 36: A labelled diagram of an ellipse.

In this example script, we illustrate the text of Section 10.13 by using Pyxplot's `ellipse` command, together with arrows and text labels, to produce a labelled diagram of an ellipse. We label the semi-major axis a, the semi-minor axis b, the semi-latus rectum L, and the distance between the centre of the ellipse and one of its foci with the length ae, where e is the eccentricity of the ellipse.

```
set multiplot ; set nodisplay

a   = 6.0              # Semi-major axis
b   = 4.0              # Semi-minor axis
e   = sqrt(1-(b/a)**2) # Eccentricity
slr = a*(1-e**2)       # Length of semi-latus rectum
fd  = a*e              # Distance of focus from center

# Draw ellipse
ellipse center 0,0 semiMajor a semiMinor b with lw 3

# Draw points at center and focus
set texthalign center ; set textvalign top
set fontsize 1.5
point at 0,0 label "Centre" with pointsize 2 plw 2
point at -fd,0 label "Focus" with pointsize 2 plw 2
```

```
# Draw arrows and dotted lines on ellipse
arrow from 0,0 to 0,b with twohead lw 2 lt 3   # Semi-minor axis
arrow from 0,0 to a,0 with twohead lw 2 lt 3   # Semi-major axis
arrow from -fd,0 to -fd,slr with tw lw 2 lt 3  # SLR
arrow from 0,0 to -fd,0 with twohead lw 2 lt 3 # Focus <-> Centre

# Label ellipse
set texthalign center ; set textvalign center
text '$ae$' at -fd/2,-0.3
text '$a$' at a/2,+0.3
text '$b$' at 0.3,b/2
set texthalign left ; set textvalign center
text '$L=a(1-e^2)$' at  0.2-fd,slr/2

# Display diagram
set display ; refresh
```



10.14 The `piechart` command

The `piechart` command produces piecharts based upon single columns of data read from data files, which are taken to indicate the sizes of the pie wedges. The `piechart` command has the following syntax:

```
piechart ('<filename>'|<function>)
     [using <using specifier>]
     [select <select specifier>]
     [index <index specifier>]
     [every <every specifier>]
     [label <auto|key|inside|outside> <label>]
     [format <format string>]
     [with <style> [<style modifier> ... ] ]
```

Immediately after the `piechart` keyword, the file (or indeed, function) from which the data is to be taken should be specified; any of the modifiers taken by the `plot` command – i.e. `using`, `index`, etc. – may be used to specify which

data from this data file should be used. The `label` modifier should be used to specify how a name for each pie wedge should be drawn from the data file, and has a similar syntax to the equivalent modifier in the `plot` command, except that the name string may be prefixed by a keyword to specify how the pie wedge names should be positioned. Four options are available:

- `auto` – specifies that the `inside` positioning mode should be used on wide pie wedges, and the `outside` positioning mode should be used on narrow pie wedges. **[default]**
- `key` – specifies that all of the labels should be arranged in a vertical list to the right-hand side of the piechart.
- `inside` – specifies that the labels should be placed within the pie wedges themselves.
- `outside` – specifies that the labels should be arranged around the circumference of the pie chart.

Having specified a name for each wedge using the `label` modifier, the `format` modifier determines the final text which is printed along side each wedge. For example, a wedge with name 'Europe' might be labelled as '27% Europe', applying the default format string:

```
"%.1d\%% %s"%(percentage,label)
```

Three variables may be used in format strings: `label` contains the name of the wedge as specified by the `label` modifier, `percentage` contains the numerical percentage size of the wedge, and `wedgesize` contains the absolute unnormalised size of the wedge, as read from the input data file, before the sizes were renormalised to sum to 100%.

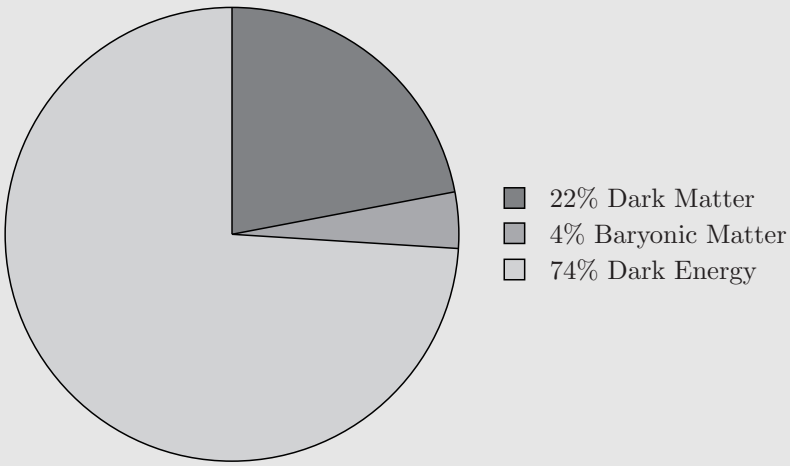
The `with` modifier may be followed by the keywords `color`, `linewidth`, `style`, which all apply to the lines drawn around the circumference of the piechart and between its wedges. The fill color of the wedges themselves are taken sequentially from the current palette, as set by the `set palette` command. Note that Pyxplot's default palette is optimised more for producing plots with datasets in different and distinct colors than for producing piecharts in aesthetically pleasing shades, where a little more subtly may be desirable. A suitable call to the `set palette` command is highly recommended before the `piechart` command is used.

As with the `plot` command, the position and size of the piechart are governed by the `set origin` and `set size` commands. The former determines where the centre of the piechart is positioned; the latter determines its diameter.

Example 37: A piechart of the composition of the Universe.

In this example, we use Pyxplot's `piechart` command to produce a diagram of the composition of the Universe, showing that of the mass in the Universe, only 4% is in the form of the baryonic matter; of the rest, 22% is in the form of dark matter and 74% in the form of dark energy:

```
set palette gray40, gray60, gray80
set width 6
piechart '--' using $1 label key "%s"%($2)
0.22 Dark~Matter
0.04 Baryonic~Matter
0.74 Dark~Energy
END
```



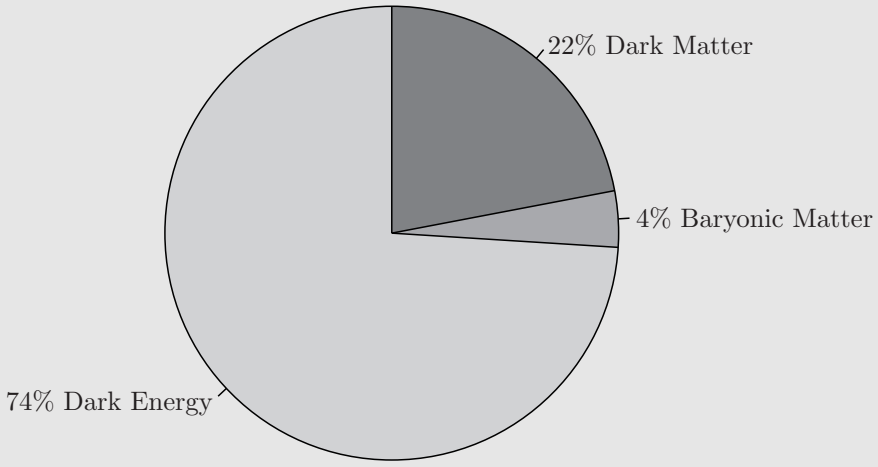
Below, we show the change produced by replacing the line

```
piechart '--' using $1 label key "%s"%($2)
```

with

```
piechart '--' using $1 label auto "%s"%($2)
```

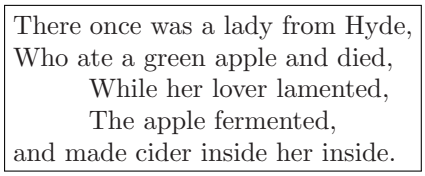
Note that the labels on the piechart are placed either within the pie, in the cases of large wedges, and around the edge of the pie for those wedges which are too narrow for this.



10.15 LaTeX and Pyxplot

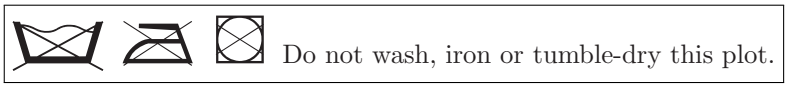
The `text` command can straightforwardly be used to render simple one-line latex strings, but sometimes the need arises to place more substantial blocks of text onto a plot. For this purpose, it can be useful to use the latex `parbox` or `minipage` environments[3]. For example:

```
text r'\parbox[t]{6cm}{\setlength{\parindent}{1cm} \
\noindent There once was a lady from Hyde, \newline \
Who ate a green apple and died, \newline \
\indent While her lover lamented, \newline \
\indent The apple fermented, \newline \
and made cider inside her inside.}'
```



If unusual mathematical symbols are required, for example those in the `amsmath` package, such a package can be loaded using the `set preamble` command. For example:

```
set preamble \usepackage{marvosym}
text r"{\Huge\Dontwash\ \NoIroning\ \NoTumbler}$\;$ Do not \
wash, iron or tumble-dry this plot."
```



[3]Remember, any valid latex string can be passed to the `text` command and `set label` command.

Part III

Reference manual

Chapter 11

Command reference

This chapter contains an alphabetically ordered list of all of Pyxplot's commands. The syntax of each is specified in a variant of Backus-Naur notation, in which angle brackets `<>` are used to indicate replaceable tokens, parentheses `()` are used to indicate mutually-exclusive options which are separated by vertical lines `|`, square brackets `[]` are used to indicate optional items, and braces {} are used to indicate items which may be repeated. Dots `...` are used to indicate arbitrary strings of text.

Replaceable tokens labelled `<length>` may be specified either as a number with physical dimensions of length, e.g. `2*unit(m)`, or as a dimensionless number taken as a number of centimeters. Replaceable tokens labelled `<angle>` may be specified either with physical dimensions of angle, e.g. `0.25*unit(rev)`, or as a dimensionless number of degrees, e.g. `90`.

Replaceable tokens labelled `<vector>` represent a physical position on the vector-graphics canvas, and can be specified either as two comma-separated co-ordinates, or as a two-component vector. In either case, the co-ordinates may either have physical demensions of length, or be a dimensionless number of centimeters.

Replaceable tokens labelled `<graph_vector>` are similar, but represent a position on a graph. They may be specified either as comma-separated co-ordinates, or as a vector object. In either case, there may be either two or three components, although the third component will be ignored except on three-dimensional plots. The components should share the physical units of the axes they are plotted against.

In flow control commands, tokens labelled `<code>` should be replaced by a series of Pyxplot commands enclosed by braces {}. The closing brace must be placed on a new line.

Where braces {} are used to indicate items which may be repeated, commas or semicolons are often used to separate items. This is specified in the text below the syntax specification.

Where keywords differ between US and British English, both variants are accepted. For example, `color` may be spelt `colour`, `gray` may be spelt `grey`, etc.

11.1 ?

```
? [ <topic> { <sub-topic> } ]
```

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

11.2 !

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

Shell commands can be executed within Pyxplot by prefixing them with pling (!) characters, as in the example:

```
!mkdir foo
```

As an alternative, back-quotes (`) can be used to substitute the output of a shell command into a Pyxplot command, as in the 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`''
```

11.3 arc

```
arc [ item <id> ] [at] <vector> radius <length>
    from <angle> to <angle> [ with { <option> } ]
```

Arcs (curves with constant radius of curvature, that is, segments of circles) may be drawn on multiplot canvases using the `arc` command. The `at` modifier specifies the coordinates of the center of curvature, from which all points on the arc are at the distance given following the `radius` modifier. The angles `start` and `finish`, measured clockwise from the vertical, control where the arc begins and ends. For example, the command

```
arc at 0,0 radius 2 from 90 to 270
```

would draw a semi-circle beneath the line $x = 0$, centered on the origin with radius 2 cm. The usual style modifiers for lines may be passed after the keyword `with`; if the `fillcolor` modifier is specified then the arc will be filled to form a pie-chart slice.

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.4 arrow

```
arrow [ item <id> ] [from] <vector> to <vector>
                  [ with { <option> } ]
```

Arrows may be drawn on multiplot canvases using the `arrow` command. The style of the arrows produced may be specified by following the `with` modifier by one of the style keywords `nohead`, `head` (default) or `twohead`. In addition, keywords such as `color`, `linewidth` and `linetype` have the same syntax and meaning following the keyword `with` as in the `plot` command. The following example would draw a bidirectional blue arrow:

```
arrow from x1,y1 to x2,y2 with twohead linetype 2 color blue
```

The `arrow` command has a twin, the `line` command, which has the same syntax, but uses the default arrow style of `nohead`, producing short line segments.

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.5 assert

```
assert ( <expression> | version ( >= | < ) <version> )
       [ <error message> ]
```

The `assert` command can be used to assert that a logical expression, such as `x>0`, is true. An error is reported if the expression is false, and optionally a string can be supplied to provide a more informative error message to the user:

```
assert x>0
assert y<0 "y must be less than zero."
```

The `assert` command can also be used to test the version number of Pyxplot. It is possible to test either that the version is newer than or equal to a specific version, using the `>=` operator, or that it is older than a specific version, using the `<` operator, as demonstrated in the following examples:

```
assert version >= 0.8.2
assert version <  0.8  "This script is designed for Pyxplot 0.7"
```

11.6 box

```
box [ item <id> ] at <vector> width <length> height <length>
         [ rotate <angle> ] [ with { <option> } ]

box [ item <id> ] from <vector> to <vector>
         [ rotate <angle> ] [ with { <option> } ]
```

The `box` command is used to draw and fill rectangular boxes on multiplot canvases. The position of each box may be specified in one of two ways. In the first, the coordinates of one corner of the box are specified, along with its width and height. If both the width and the height are positive then the coordinates are taken to be those of the bottom left-hand corner of the box; other corners may be specified if the supplied width and/or height are negative. If a rotation angle is specified then the box is rotated about the specified corner. The `with` modifier allows the style of the box to be specified using similar options to those accepted by the `plot` command.

The second syntax allows two pairs of coordinates to be specified. Pyxplot will then draw a rectangular box with opposing corners at the specified locations. If an angle is specified the box will be rotated about its center. Hence the following two commands both draw a square box centered on the origin:

```
box from -1, -1 to 1,1
box at 1, -1 width -2 height 2
```

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.7 break

```
break [ <loopname> ]
```

The `break` command terminates execution of `do`, `while`, `for` and `foreach` loops in an analogous manner to the `break` statement in the C programming language. Execution resumes at the statement following the end of the loop. For example, the following loop would only print the numbers 1 and 2:

```
for i = 1 to 10
 {
  print i
  if (i==2)
   {
    break
   }
 }
```

If several loops are nested, the `break` statement only acts on the innermost loop. If the `break` statement is encountered outside of any loop structure, an error results. Optionally, the `for`, `foreach`, `do` and `while` commands may be supplied with a name for the loop, prefixed by the word `loopname`, as in the examples:

```
for i=0 to 4 loopname iloop

foreach i in "*.dat" loopname DatafileLoop
```

When loops are given such names, the `break` statement may be followed by the name of the loop whose iteration is to be broken, allowing it to act upon loops other than the innermost one.

See also the `continue` command.

11.8 call

```
call <expression>
```

The `call` command evaluates a function or subroutine call, and discards the result. Whereas entering `f(x)` on the commandline alone will print the result of the function call, `call f(x)` quietly discards the function evaluation.

11.9 cd

```
cd <directory>
```

Pyxplot's `cd` command is very similar to the shell `cd` command; it changes the current working directory. The following example would enter the subdirectory `foo`:

```
cd foo
```

11.10 circle

```
circle [ item <id> ] [at] <vector> radius <length>
       [ with { <option> } ]
```

The `circle` command is used to draw circles on multiplot canvases. The coordinates of the circle's center and its radius are specified. The `with` modifier allows the style of the circle to be specified using similar options to those accepted by the `plot` command. The example

```
circle at 2,2 radius 1 with color red fillcolor blue
```

would draw a red circle of unit radius filled in blue, centered 2 cm above and to the right of the origin.

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.11 clear

```
clear
```

In multiplot mode, the `clear` command removes all plots, arrows and text objects from the working multiplot canvas. Outside of multiplot mode, it is not especially useful; it removes the current plot to leave a blank canvas. The `clear` command should not be followed by any parameters.

11.12 continue

```
continue [ <loopname> ]
```

The `continue` command terminates execution of the current iteration of `for`, `foreach`, `do` and `while` loops in an analogous manner to the `continue` statement in the C programming language. Execution resumes at the first statement of the next iteration of the loop, or at the first statement following the end of the loop in the case of the last iteration of the loop. For example, the following script will not print the number 2:

```
for i = 0 to 5
 {
  if (i==2)
   {
    continue
   }
  print i
 }
```

If several loops are nested, the `continue` statement only acts on the innermost loop. If the `continue` statement is encountered outside of any loop structure, an error results. Optionally, the `for`, `foreach`, `do` and `while` statements may be supplied with a name for the loop, prefixed by the word `loopname`, as in the examples:

```
for i=0 to 4 loopname iloop

foreach i in "*.dat" loopname DatafileLoop
```

When loops are given such names, the `continue` statement may be followed by the name of the loop whose iteration is to be broken, allowing it to act upon loops other than the innermost one.

See also the `break` command.

11.13 delete

```
delete { <item number> }
```

The `delete` command removes vector graphics objects such as plots, arrows or text items from the current multiplot canvas. All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. The items to be deleted should be identified using a comma-separated list of their identification numbers. The example

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

would remove item numbers 1, 2 and 3.

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

11.14 do

```
do [ loopname <loopname> ]
 <code>
while <condition>
```

The `do` command executes a block of commands repeatedly, checking the condition given in the `while` clause at the end of each iteration. If the condition is true then the loop executes again. This is similar to a `while` loop, except that the contents of a `do` loop are *a*lways executed at least once. The following example prints the numbers 1, 2 and 3:

```
i=1
do
 {
  print i
  i = i + 1
 } while (i < 4)
```

Note that there must always be a newline following the opening brace after the `do` command, and the while clause must always be on the same line as the closing brace.

11.15 ellipse

Ellipses may be drawn on multiplot canvases using the `ellipse` command. The shape of the ellipse may be specified in many different ways, by specifying

(i) the position of two corners of the smallest rectangle which can enclose the ellipse when its major axis is horizontal, together with an optional counter-clockwise rotation angle, applied about the center of the ellipse. For example:

```
ellipse from 0,0 to 4,1 rot 70
```

(ii) the position of both the center and one of the foci of the ellipse, together with any one of the following additional pieces of information: the ellipse's major axis length, its semi-major axis length, its minor axis length, its semi-minor axis length, its eccentricity, its latus rectum, or its semi-latus rectum. For example:

```
ellipse focus 0,0 center 2,2 majoraxis 4
ellipse focus 0,0 center 2,2 minoraxis 4
ellipse focus 0,0 center 2,2 ecc 0.5
ellipse focus 0,0 center 2,2 LatusRectum 6
ellipse focus 0,0 center 2,2 slr 3
```

(iii) the position of either the center or one of the foci of the ellipse, together with any two of the following additional pieces of information: the ellipse's major axis length, its semi-major axis length, its minor axis length,

its semi-minor axis length, its eccentricity, its latus rectum, or its semi-latus rectum. An optional counter-clockwise rotation angle may also be specified, applied about either the center or one of the foci of the ellipse, whichever is specified. If no rotation angle is given, then the major axis of the ellipse is horizontal. For example:

```
ellipse center 0,0 majoraxis 4 minoraxis 4
```

Optionally, an arc of an ellipse may be drawn by adding the following modified:

```
arc from <angle> to <angle>
```

The line type, line width, and color of line with which the outlines of ellipses are drawn may be specified after the keyword `with`, as in the `box` and `circle` commands above. Likewise, ellipses may be filled in the same manner.

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.16 else

The `else` statement is described in the entry for the `if` statement, of which it forms part.

11.17 eps

```
eps [ item <id> ] <filename> [ at <vector> ] [rotate <angle>]
                             [width <length>] [height <length>]
                                          [ clip ] [ calcbbox ]
```

The `eps` command allows Encapsulated PostScript (EPS) images to be inserted onto multiplot canvases. The `at` modifier can be used to specify where the image should be placed on the vector graphics canvas; if it is not, then the image is placed at the origin. The settings `texthalign` and `textvalign` determined how the image is aligned relatively to this reference point – for example, whether its bottom left corner or its center is placed at the reference point.

The `rotate` modifier can be used to rotate the image by any angle, measured in degrees counter-clockwise. The `width` or `height` modifiers can be used to specify the width or height with which the image should be rendered; both should be specified in centimeters. If neither is specified then the image will be rendered with the native dimensions specified within the PostScript. The `eps` command is often useful in multiplot mode, allowing PostScript images to be combined with plots, text labels, etc.

The `clip` modifier causes Pyxplot to clip an eps image to its stated bounding box. The `calcbbox` modifier causes Pyxplot to ignore the bounding box stated

in the eps file and calculate its own when working out how to scale the image to the specified width and height.

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.18 exec

```
exec <command>
```

The `exec` command can be used to execute Pyxplot commands contained within string variables, as in the following example:

```
terminal="eps"
exec "set terminal %s"%(terminal)
```

It can also be used to write obfuscated Pyxplot scripts, and its use should be minimized wherever possible.

11.19 exit

```
exit
```

The `exit` command can be used to quit Pyxplot. If multiple command files, or a mixture of command files and interactive sessions, were specified on Pyxplot's 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 using 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 the script.

11.20 fft

```
fft { <range> } <function name>()
    of ( <filename> | <function name>() )
    [ using { <expression> } ]

ifft { <range> } <function name>()
    of ( <filename> | <function name>() )
    [ using { <expression> } ]
```

The `fft` command calculates Fourier transforms of data files or functions. Transforms can be performed on datasets with arbitrary numbers of dimensions. To transform an algebraic expression with n degrees of freedom, it must be wrapped in a function of the form $f(i_2, i_2, \ldots, i_n)$. To transform an n-dimensional dataset stored in a data file, the samples must be arranged on a regular linearly-spaced grid and stored in row-major order. For each dimension

of the transform, a range specification must be provided to the `fft` command in the form

```
[ <minimum> : <maximum> : <step> ]
```

When data from a data file is being transformed, the specified range(s) must precisely match those of the samples read from the file; the first n columns of data should contain the values of the n real-space coordinates, and the $n+1$th column should contain the data to be transformed. After the range(s), a function name should be provided for the output transform: a function of n arguments with this name will be generated to represent the transformed data. Note that this function is in general complex – i.e. it has a non-zero imaginary component. Complex numerics can be enabled using the `set numerics complex` command and the `fft` command is of little use without doing so. The `using`, `index`, `every` and `select` modifiers can be used to specify how data will be sampled from the input function or data file in an analogous manner to how they are used in the `plot` command.

The `ifft` command calculates inverse Fourier transforms; it has the same syntax as the `fft` command.

11.21 fit

```
fit [ { <range> } ] <function name>() [ withouterrors ]
    ( <filename> | { <expression> } | { <vector obj> } )
    [ index <value> ] [ using { <expression> } ]
    via { <variable> }
```

The `fit` command can be used to fit arbitrary functional forms to data points read from files. It can be used to produce best-fit lines for datasets or to determine gradients and other mathematical properties of data by looking at the parameters associated with the best-fitting functional form. The following simple example fits a straight line to data in a file called `data.dat`:

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

The first line specifies the functional form which is to be used. The coefficients within this function, `a` and `b`, which are to be varied during the fitting process are listed after the keyword `via` in the `fit` command. The modifiers `index`, `every`, `select` and `using` have the same meanings in the `fit` command as in the `plot` command. When fitting a function of n variables, at least $n+1$ columns (or rows – see Section 3.9.1) of data 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 arguments 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 data points which are being used to constrain the fit.

As the `fit` command works, it displays statistics including the best-fit values of each of the fitting parameters, the uncertainties in each of them, and

the covariance matrix. These can be useful for analysing the security of the fit achieved, but calculating the uncertainties in the best-fit parameters and the covariance matrix can be time consuming, especially when many parameters are being fitted simultaneously. The optional keyword `withouterrors` can be included immediately before the filename of the data file to be fitted to substantially speed up cases where this information is not required.

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() 'data.dat' index 1 using 2:3 via a,b
```

More details can be found in Section 5.6.

11.22 for

```
for <variable> = <start> to <end> [step <step>]
                            [loopname <loopname>]
  <code>

for (<initialise>; <criterion>; <step>)
  <code>
```

The `for` command executes a set of commands repeatedly. Pyxplot allows `for` loops to follow either the syntax of the BASIC programming language, or the C syntax.

In the BASIC variant, a specified variable takes a different value on each iteration. The variable takes the value `start` on the first iteration, and increases by a fixed value `step` on each iteration; `step` may be negative if `end` $<$ `start`. If `step` is not specified then a value of unity is assumed. The loop terminates when the variable exceeds `end`. The following example prints the squares of the first five natural numbers:

```
for i = 1 to 5
 {
  print i**2
 }
```

In the C variant, three expressions are provided, which are evaluated (a) when the loop initialises, (b) as a boolean test of whether the loop should continue iterating, and (c) on each loop to increment/decrement variables as required. For example:

```
for (i=1,j=1; i<=256; i*=2,j++) { print "%3d %3d"%(j,i); }
```

11.23 foreach

```
foreach <variable> in ( <filename wildcard> |
                        <list> )
                      [ loopname <loopname> ]
  <code>
```

The `foreach` command can be used to run a block of commands repeatedly, once for each item in a list. The list of items can be specified in one of two ways. In the first case, a set of filenames or filename wildcards is supplied, and the `foreach` loop iterates once for each supplied filename, with a string variable set to each filename in succession. For example, the following loop would plot the data in the set of files whose names end with `.dat`:

```
plot     # Create blank plot
foreach file in *.dat
 {
  replot file with lines
 }
```

The second form of the command takes a list of string or numerical values provided explicitly by the user, and the `foreach` loop iterates once for each value, with a variable set to each value in succession. For example, the following script would plot normal distributions of three different widths:

```
plot     # Create blank plot
foreach sigma in (1, 2, 3)
 {
  replot 1/sigma*exp(-x**2/(2*sigma**2))
 }
```

11.24 foreach datum

```
foreach { <variable> } in [ { <range> } ]
    ( <filename> | { <expression> } | { <vector obj> } )
    [ every { <expression> } ]
    [ index <value> ] [ select <expression> ]
    [ using { <expression> } ]
  <code>
```

The `foreach datum` command executes a series of commands for each data pointread from a data fileor function. It takes a similar set of modifiers to the `plot` command; the list of variables to be read from the supplied data on each iteration should be comma separated.

11.25 global

```
global { <variable name> }
```

The `global` command command may be used within subroutines, which have their own private variable namespaces, to specify that the named variables should refer to the global namespace. If multiple variables are specified, their names should be comma separated.

11.26 help

```
help [ <topic> { <sub-topic> } ]
```

The `help` command provides an hierarchical source of information which is supplementary to this Users' Guide. To obtain information on any particular topic, type `help` followed by the name of the topic, as in the following example

```
help commands
```

which provides information on Pyxplot's commands. Some topics have sub-topics; these are listed at the end of each help page. To view them, add further words to the end of the help request, as in the example:

```
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 data files should take can be found under `datafile`. Other categories are self-explanatory.

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

11.27 histogram

```
histogram [ <range> ] <function name>()
     ( <filename> | { <expression> } | { <vector obj> } )
     [ every { <expression> } ]
     [ index <value> ]
     [ select <expression> ]
     [ using { <expression> } ]
     ( [ binwidth <value> ] [ binorigin <value> ] |
       [ bins (x1, x2, ...) ] )
```

The `histogram` command takes a single column of data from a file and produces a function that represents the frequency distribution of the supplied data values. The output function consists of a series of discrete intervals which we term *bins*. Within each interval the output function has a constant value, determined such that the area under each interval – i.e. the integral of the function over each interval – is equal to the number of datapoints found within that interval. The following simple example

```
histogram f() 'input.dat'
```

produces a frequency distribution of the data values found in the first column of the file `input.dat`, which it stores in the function $f(x)$. The value of this function at any given point is equal to the number of items in the bin at that point, divided by the width of the bins used. If the input datapoints are not dimensionless then the output frequency distribution adopts appropriate units, thus a histogram of data with units of length has units of one over length.

The number and arrangement of bins used by the `histogram` command can be controlled by means of various modifiers. The `binwidth` modifier sets the width of the bins used. The `binorigin` modifier controls where their boundaries lie; the `histogram` command selects a system of bins which, if extended to infinity in both directions, would put a bin boundary at the value specified in the `binorigin` modifier. Thus, if `binorigin 0.1` were specified, together with a bin width of 20, bin boundaries might lie at 20.1, 40.1, 60.1, and so on. Alternatively global defaults for the bin width and the bin origin can be specified using the `set binwidth` and `set binorigin` commands respectively. The example

```
histogram h() 'input.dat' binorigin 0.5 binwidth 2
```

would bin data into bins between 0.5 and 2.5, between 2.5 and 4.5, and so forth.

Alternatively the set of bins to be used can be controlled more precisely using the `bins` modifier, which allows an arbitrary set of bins to be specified. The example

```
histogram g() 'input.dat' bins (1, 2, 4)
```

would bin the data into two bins, $x = 1 \to 2$ and $x = 2 \to 4$.

A range can be supplied immediately following the `histogram` command, using the same syntax as in the `plot` and `fit` commands; if such a range is supplied, only points that fall within that range will be binned. In the same way as in the `plot` command, the `index`, `every`, `using` and `select` modifiers can be used to specify which subsets of a data file should be used.

Two points about the `histogram` command are worthy of note. First, although histograms are similar to bar charts, they are not the same. A bar chart conventionally has the height of each bar equal to the number of points that it represents, whereas a histogram is a continuous function in which the area underneath each interval is equal to the number of points within it. Thus, to produce a bar chart using the `histogram` command, the end result should be multiplied by the bin width used.

Second, if the function produced by the `histogram` command is plotted using the `plot` command, samples are automatically taken not at evenly spaced intervals along the ordinate axis, but at the centers of each bin. If the `boxes` plot style is used, the box boundaries are be conveniently drawn to coincide with the bins into which the data were sorted.

11.28 history

```
history [ <number of items> ]
```

The `history` command prints a list of the most recently executed commands on the terminal. The optional parameter, `N`, if supplied, causes only the latest N commands to be displayed.

11.29 if

```
if <criterion> <code>
{ else if <criterion> <code> }
[ else                <code> ]
```

The `if` command allows conditional execution of blocks of commands. The code enclosed in braces following the `if` statement is executed if, and only if, the `criterion` is satisfied. An arbitrary number of subsequent `else if` statements can optionally follow the initial `if` statement; these have their own criteria for execution which are only considered if all of the previous criteria have tested false – i.e. if none of the previous command blocks have been executed. A final optional `else` statement can be provided; the block of commands which follows it are executed only if none of the preceding criteria have tested true. The following example illustrates a chain of `else if` clauses:

```
if (x==2)
 {
  print "x is two!"
 } else if (x==3) {
  print "x is three!"
 } else if (x>3) {
  print "x is greater than three!"
 } else {
  x=2
  print "x didn't used to be two, but it is now!"
 }
```

11.30 ifft

```
ifft { <range> } <function name>()
    of ( <filename> | <function name>() )
    [using { <expression> } ]
```

See `fft`.

11.31 image

```
image [ item <id> ] <filename> [ at <vector> ]
         [ rotate <angle> ] [ width <length> ]
         [ height <length>] [ smooth ]
         [ notransparent ] [ transparent rgb <r>:<g>:<b> ]
```

The `image` command allows graphical images to be inserted onto the current multiplot canvas from files on disk. Input graphical images may be in bitmap, gif, jpeg or png formats; the file type of each image is automatically detected. The `at` modifier can be used to specify where the image should be placed on the vector graphics canvas; if it is not, then the image is placed at the origin. The settings `texthalign` and `textvalign` determined how the image is aligned

relatively to this reference point – for example, whether its bottom left corner or its center is placed at the reference point.

The `rotate` modifier can be used to rotate images by any angle, measured in degrees counter-clockwise. The `width` or `height` modifiers can be used to specify the width or height with which images should be rendered; both should be specified in centimeters. If neither is specified then images are rendered with the native dimensions specified within the metadata present in the image file (if any). If both are specified, then the aspect ratio of the image is changed.

The keyword `smooth` may optionally be supplied to cause the pixels of images to be interpolated[1]. Images which include transparency are supported. The optional keyword `notransparent` may be supplied to cause transparent regions to be filled with the image's default background color. Alternatively, an RGB color may be specified in the form `rgb<r>:<g>:<b>` after the keyword `transparent` to cause that particular color to become transparent; the three components of the RGB color should be in the range 0 to 255.

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.32 interpolate

```
interpolate ( akima | linear | loglinear | polynomial |
              spline | stepwise |
              2d [ ( bmp_r | bmp_g | bmp_b ) ] )
            [ <range specification> ] <function name>()
            <filename>
            [ every { <expression> } ]
            [ index <value> ]
            [ select <expression> ]
            [ using { <expression> } ]
```

The `interpolate` command can be used to generate a special function within Pyxplot's mathematical environment which interpolates a set of data points supplied from a data file. Either one- or two-dimensional interpolation is possible.

In the case of one-dimensional interpolation, various different types of interpolation are supported: linear interpolation, power law interpolation, polynomial interpolation, cubic spline interpolation and akima spline interpolation. Stepwise interpolation returns the value of the datapoint nearest to the requested point in argument space. The use of polynomial interpolation with large datasets is strongly discouraged, as polynomial fits tend to show severe oscillations between data points. Except in the case of stepwise interpolation, extrapolation is not permitted; if an attempt is made to evaluate an interpolated function beyond the limits of the data points which it interpolates, Pyxplot returns an error or value of not-a-number.

[1] Many commonly-used PostScript display engines, including Ghostscript, do not support this functionality.

In the case of two-dimensional interpolation, the type of interpolation to be used is set using the `interpolate` modifier to the `set samples` command, and may be changed at any time after the interpolation function has been created. The options available are nearest neighbor interpolation – which is the two-dimensional equivalent of stepwise interpolation, inverse square interpolation – which returns a weighted average of the supplied data points, using the inverse squares of their distances from the requested point in argument space as weights, and Monaghan Lattanzio interpolation, which uses the weighting function (Monaghan & Lattanzio 1985)

$$\begin{aligned} w(x) &= 1 - {}^3\!/_2 v^2 + {}^3\!/_4 v^3 \quad \text{for } 0 \le v \le 1 \\ &= {}^1\!/_4 (2-v)^3 \quad \text{for } 1 \le v \le 2 \end{aligned}$$

where $v = r/h$ for $h = \sqrt{A/n}$, A is the product $(x_{\max} - x_{\min})(y_{\max} - y_{\min})$ and n is the number of input datapoints. These are selected as follows:

```
set samples interpolate nearestNeighbor
set samples interpolate inverseSquare
set samples interpolate monaghanLattanzio
```

Finally, data can be imported from graphical images in bitmap (`.bmp`) format to produce a function of two arguments returning a value in the range $0 \to 1$ which represents the data in one of the image's three color channels. The two arguments are the horizontal and vertical position within the bitmap image, as measured in pixels.

A very common application of the `interpolate` command is to perform arithmetic functions such as addition or subtraction on datasets which are not sampled at the same abscissa values. The following example would plot the difference between two such datasets:

```
interpolate linear f() 'data1.dat'
interpolate linear g() 'data2.dat'
plot [min:max] f(x)-g(x)
```

Note that it is advisable to supply a range to the `plot` command in this example: because the two datasets have been turned into continuous functions, the `plot` command has to guess a range over which to plot them unless one is explicitly supplied.

The `spline` command is an alias for `interpolate spline`; the following two statements are equivalent:

```
spline f() 'data1.dat'
interpolate spline f() 'data1.dat'
```

11.33 jpeg

```
jpeg [ item <id> ] <filename> [ at <vector> ]
         [ rotate <angle> ] [ width <length> ]
         [ height <length>] [ smooth ]
         [ notransparent ] [ transparent rgb <r>:<g>:<b> ]
```

See `image`.

11.34 let

```
let <variable name> = <value>
```

The `let` command sets the named variable to equal `value`.

11.35 list

```
list
```

The `list` command returns a list of all of the items on the current multiplot canvas, giving their identification numbers and the commands used to produce them. The following is an example of the output produced:

```
pyxplot> list

# ID   Command
    1  plot item 1 f(x) using columns
    2  [deleted] text item 2 "Figure 1: A plot of f(x)" at 0,0 rotate 0 gap 0
    3  text item 3 "Figure 1: A plot of $f(x)$" at 0,0 rotate 0 gap 0
```

In this example, the user has plotted a graph of $f(x)$ and added a caption to it. The `ID` column lists the reference numbers of each multiplot item. Item `1` has been deleted.

11.36 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 terminal. Filename wildcard can be supplied, in which case *all* command files matching the given wildcard are executed, as in the example:

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

11.37 local

```
local { <variable name> }
```

The `global` command command may be used within subroutines, which have their own private variable namespaces. It specifies that the named variables should, from now on, refer to the local namespace, even if the `global` command command has previously been used to reference the global namespace. If multiple variables are specified, their names should be comma separated.

11.38 maximize

```
maximize <expression> via { <variable> }
```

The `maximize` command can be used to find the maxima of algebraic expressions. A single algebraic expression should be supplied for optimisation, together with a comma-separated list of the variables with respect to which it should be optimised. In the following example, a maximum of the sinusoidal function $\cos(x)$ is sought:

```
pyxplot> set numerics real
pyxplot> x=0.1
pyxplot> maximize cos(x) via x
pyxplot> print x/pi
-7.15135259e-52
```

Note that this particular example doesn't work when complex arithmetic is enabled, since $\cos(x)$ diverges to ∞ at $x = \infty i$.

Various caveats apply the `maximize` command, as well as to the `minimize` and `solve` commands. All of these commands operate by searching numerically for optimal sets of input parameters to meet the criteria set by the user. As with all numerical algorithms, there is no guarantee that the *locally* optimum solutions returned are the *globally* optimum solutions. It is always advisable to double-check that the answers returned agree with common sense.

These commands can often find solutions to equations when these solutions are either very large or very small, but they usually work best when the solution they are looking for is roughly of order unity. Pyxplot does have mechanisms which attempt to correct cases where the supplied initial guess turns out to be many orders of magnitude different from the true solution, but it cannot be guaranteed not to wildly overshoot and produce unexpected results in such cases. To reiterate, it is always advisable to double-check that the answers returned agree with common sense.

11.39 minimize

```
minimize <expression> via { <variable> }
```

The `minimize` command can be used to find the minima of algebraic expressions. A single algebraic expression should be supplied for optimisation, together with a comma-separated list of the variables with respect to which it should be optimised. In the following example, a minimum of the sinusoidal function $\cos(x)$ is sought:

```
pyxplot> set numerics real
pyxplot> x=0.1
pyxplot> minimize cos(x) via x
pyxplot> print x/pi
1
```

Note that this particular example doesn't work when complex arithmetic is enabled, since $\cos(x)$ diverges to $-\infty$ at $x = \pi + \infty i$.

Various caveats apply the `minimize` command, as well as to the `maximize` and `solve` commands. All of these commands operate by searching numerically for optimal sets of input parameters to meet the criteria set by the user. As with all numerical algorithms, there is no guarantee that the *locally* optimum solutions returned are the *globally* optimum solutions. It is always advisable to double-check that the answers returned agree with common sense.

These commands can often find solutions to equations when these solutions are either very large or very small, but they usually work best when the solution they are looking for is roughly of order unity. Pyxplot does have mechanisms which attempt to correct cases where the supplied initial guess turns out to be many orders of magnitude different from the true solution, but it cannot be guaranteed not to wildly overshoot and produce unexpected results in such cases. To reiterate, it is always advisable to double-check that the answers returned agree with common sense.

11.40 move

```
move <item number> to <vector> [ rotate <angle> ]
```

The `move` command allows vector graphics objects to be moved around on the current multiplot canvas. All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. The item to be moved should be specified using its identification number. The example

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

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

11.41 piechart

11.42 plot

```
plot [ 3d ] [ item <id> ] [ { <range> } ]
     ( <filename> | { <expression> } | { <vector obj> } )
     [ axes <axes>] [ every { <expression> } ]
     [ index <value>] [ select <expression> ]
     [ label <string expression> ]
     [ title <string> ] [ using { <expression> } ]
     [ with { <option> } ]
```

The `plot` command is used to produce graphs. The following simple example would 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. An example of this syntax would be:

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

which would plot the function $\sin(x)$ between $-\pi$ and π.

Data files may also be plotted as well as functions, in which case the filename of the data file to be plotted should be enclosed in either single or double quotation marks. An example of this syntax would be:

```
plot 'data.dat' with points
```

which would plot data from the file `data.dat`. Section 3.8 provides further details of the format that input data files should take and how Pyxplot may be directed to plot only certain portions of data files.

Multiple datasets can be plotted on a single graph by specifying them in a comma-separated list, as in the example:

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

If the `3d` modifier is supplied to the `plot` command, then a three-dimensional plot is produced; many plot styles then take additional columns of data to signify the positions of datapoints along the z-axis. This is described further in Chapter 8. The angle from which three-dimensional plots are viewed can be set using the `set view` command.

11.42.1 axes

The `axes` modifier may be used to specify which axes data should be plotted against when plots have multiple parallel axes – for example, if a plot has an `x`-axis along its lower edge and an `x2`-axis along its upper edge. The following example would plot data using the `x2`-axis as the ordinate axis and the `y`-axis as the abscissa axis:

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

It is also possible to use the `axes` modifier to specify that a vertical ordinate axis and a horizontal abscissa axis should be used; the following example would plot data using the `y`-axis as the ordinate axis and the `x`-axis as the abscissa axis:

```
plot sin(x) axes yx
```

11.42.2 label

The `label` modifier to the `plot` command may be used to render text labels next to datapoints, as in the following examples:

```
set samples 8
plot [2:5] x**2 label "$x=%.2f$"%($1) with points

plot 'datafile' using 1:2 label "%s"%($3)
```

Note that if a particular column of a data file contains strings which are to be used as labels, as in the second example above, syntax such as `"%s"%($3)` must be used to explicitly read the data as strings rather than as numerical quantities. As Pyxplot treats any whitespace as separating columns of data, such labels cannot contain spaces, though latex's $\sim$ character can be used to achieve a space.

Datapoints can be labelled when plotted in any of the following plot styles: `arrows` (and similar styles), `dots`, `errorbars` (and similar styles), `errorrange` (and similar styles), `impulses`, `linespoints`, `lowerlimits`, `points`, `stars` and `upperlimits`. It is not possible to label datapoints plotted in other styles. Labels are rendered in the same color as the datapoints with which they are associated.

11.42.3 title

By default, Pyxplot generates its own entry in the legend of a plot for each dataset plotted. This default behaviour can be overridden using the `title` modifier. The following example labels a dataset as 'Dataset 1':

```
plot sin(x) title 'Dataset 1'
```

If a blank string, i.e. `""`, is supplied, then no entry is made in the plot's legend for that dataset. The same effect can be achieved using the `notitle` modifier.

11.42.4 with

The `with` modifier controls the style in which data should be plotted. For example, the statement

```
plot "data.dat" index 1 using 4:5 with lines
```

specifies that data should be plotted using lines connecting each data pointto its neighbors. More generally, the `with` modifier can be followed by a range of settings which fine-tune the manner in which the data are displayed; for example, the statement

```
plot "data.dat" with lines linewidth 2.0
```

would use twice the default width of line.

The following is a complete list of all of Pyxplot's plot styles – i.e. all of the words which may be used in place of `lines`: `arrows_head`, `arrows_nohead`, `arrows_twohead`, `boxes`, `colormap`, `contourmap`, `dots`, `filledRegion`, `fsteps`, `histeps`, `impulses`, `lines`, `linesPoints`, `lowerLimits`, `points`, `stars`, `steps`, `surface`, `upperLimits`, `wboxes`, `xErrorBars`, `xErrorRange`, `XYErrorBars`, `xyErrorRange`, `xyzErrorBars`, `XYZErrorRange`, `xzErrorBars`, `xzErrorRange`, `yErrorBars`, `yErrorRange`, `yErrorShaded`, `yzErrorBars`, `yzErrorRange`, `zErrorBars`, `zErrorRange`. 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 the names of these plot styles, the `with` modifier can also be followed by style modifiers such as `linewidth` which alter the exact appearance of various plot styles. A complete list of these is as follows:

- `color` – used to select the color in which each dataset is to be plotted. It should be followed either by an integer, to set a color from the present palette (see Section 8.1.1), by a recognised color name, or by an object of type `color`. This modifier may also be spelt `colour`.

- `fillcolor` – used to select the color in which each dataset is filled. The color may be specified using any of the styles listed for `color`. May also be spelt `fillcolor`.

- `linetype` – used to numerically select the type of line – for example, solid, dotted, dashed, etc. – which should be used in line-based plot styles. A complete list of Pyxplot's numbered line types can be found in Chapter 18. May be abbreviated `lt`.

- `linewidth` – used to select the width of line which should be used in line-based plot styles, where unity represents the default width. May be abbreviated `lw`.

- `pointlinewidth` – used to select the width of line which should be used to stroke points in point-based plot styles, where unity represents the default width. May be abbreviated `plw`.

- `pointsize` – used to select the size of drawn points, where unity represents the default size. May be abbreviated `ps`.

- `pointtype` – used to numerically select the type of point – for example, crosses, circles, etc. – used by point-based plot styles. A complete list of Pyxplot's numbered point types can be found in Chapter 18. May be abbreviated `pt`.

Any number of these modifiers may be placed sequentially after the keyword `with`, as in the following examples:

```
plot 'datafile' using 1:2 with points pointsize 2
plot 'datafile' using 1:2 with lines color red linewidth 2
plot 'datafile' using 1:2 with lp col 1 lw 2 ps 3
```

Where modifiers take numerical values, expressions of the form `$2+1`, similar to those supplied to the `using` modifier, may be used to indicate that each datapoint should be displayed in a different style or in a different color. The following example would plot a data file with `points`, drawing the position of each point from the first two columns of the supplied data file and the size of each point from the third column:

```
plot 'datafile' using 1:2 with points pointsize $3
```

11.43 point

```
point [ item <id> ] [at] <vector> [ label <string> ]
                    [ with { <option> } ]
```

The `point` command allows a single point to be plotted on the current multiplot canvas independently of any graph. It is equivalent to plotting a data file containing a single datum and with invisible axes. If an optional `label` is specified then the text string provided is rendered next to the point. The `with` modifier allows the style of the point to be specified using similar options to those accepted by the `plot` command.

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.44 print

```
print { <expression> }
```

The `print` command displays a string or 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 textual output from a Pyxplot script. For example,

```
print a
```

would print the value of the variable `a`, and

```
print "a = %s"%(a)
```

would produce the same result in the midst of formatted text.

11.45 pwd

```
pwd
```

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

11.46 quit

```
quit
```

The `quit` command can be used to exit Pyxplot. See `exit` for more details.

11.47 rectangle

```
rectangle [ item <id> ] at <vector>
         width <length> height <length>
         [ rotate <angle> ] [ with { <option> } ]

rectangle [ item <id> ] from <vector> to <vector>
         [ rotate <angle> ] [ with { <option> } ]
```

See `box`.

11.48 refresh

```
refresh
```

The `refresh` command produces an exact copy of the latest display. It 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; use of the `set` command since the previous `plot` command has no effect on the output.

11.49 replot

```
replot [ item <id> ] ...
```

The `replot` command has the same syntax as the `plot` command and is used to add more datasets to an existing plot, or to change its axis ranges. For example, having plotted one data file using the command

```
plot 'datafile1.dat'
```

another can be plotted on the same axes using the command

```
replot 'datafile2.dat' using 1:3
```

or the ranges of the axes on the original plot can be changed using the command

```
replot [0:1][0:1]
```

The plot is also updated to reflect any changes to settings made using the `set` command. In multiplot mode, the `replot` command can likewise 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 the function $g(x)$ to it:

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

Additionally, in multiplot mode it is possible to modify any plot on the current multiplot canvas by adding an `item` modifier to the `replot` statement to specify which plot should be replotted. The following example would produce two plots, and then add an additional function to the first plot:

```
set multiplot
plot f(x)
set origin 10,0
plot g(x)
replot item 1 h(x)
```

If no `item` number is specified, then the `replot` command acts by default upon the most recent plot to have been added to the multiplot canvas.

11.50 reset

```
reset
```

The `reset` command reverts the values of all settings that have been changed with the `set` command back to their default values. It also clears the current multiplot canvas.

11.51 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 file. The filename to be used for the output should be placed in quotes, as in the example:

```
save 'foo'
```

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

11.52 set

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

The `set` command is used to configure the values of a wide range of parameters within Pyxplot which control its behaviour and the style of the output which it produces. For example:

```
set pointsize 2
```

would set the size of points plotted by Pyxplot to be twice the default. In the majority of cases, the syntax follows that above: the `set` command should be followed by a keyword which specifies which parameter should be set, followed by the value to which that parameter should be set. Those parameters which work in an on/off fashion take a different syntax along the lines of:

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

More details of the effects of each individual parameter can be found in the subsections below, which forms 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. Chapter 19 describes how commonly used settings can be saved into a configuration file.

11.52.1 arrow

```
set arrow <arrow number>
  from [ <system> ] <x>, [ <system> ] <y>, [ [ <system> ] <z> ]
  to   [ <system> ] <x>, [ <system> ] <y>, [ [ <system> ] <z> ]
  [ with { <option> } ]
```

where <system> may take any of the values
(first | second | screen | graph | axis<number>)

The set arrow command is used to add arrows to graphs. The example

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

would draw an arrow between the points $(0, 0)$ and $(1, 1)$, as measured along the x and y-axes. The tag 1 immediately following the keyword arrow is an identification number and allows arrows to be subsequently removed using the unset arrow command. By default, the coordinates are specified relative to the first horizontal and vertical axes, but they can alternatively be specified any one of several of coordinate systems. The coordinate system to be used is specified as in the example:

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

The name of the coordinate system to be used precedes the position value in that 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 centimeters from the origin of the graph. The syntax axis<n> may also be used, to use the nth horizontal or vertical 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, when placed 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. All of the line type modifiers accepted by the plot command can also be used here, as in the example:

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

11.52.2 autoscale

```
set autoscale { <axis> }
```

The set autoscale command causes Pyxplot to choose the scaling for an axis automatically based on the data and/or functions to be plotted against it. The example

```
set autoscale x1
```

would cause the range of the first horizontal axis to be scaled to fit the data. Multiple axes can be specified, as in the example

```
set autoscale x1y3
```

Note that ranges explicitly specified in a plot command will override the set autoscale command.

11.52.3 axescolor

```
set axescolor <color>
```

The setting `axescolor` changes the color used to draw graph axes. The example

```
set axescolor blue
```

would specify that graph axes should be drawn in blue. Any of the recognised color names listed in Section 19.4 can be used, or a numbered color from the present palette, or an object of type `color`.

11.52.4 axis

```
set axis <axis> [ ( visible | invisible ) ]
  [ ( top | bottom | left | right | front | back ) ]
  [ ( atzero | notatzero ) ]
  [ ( automirrored | mirrored | fullmirrored ) ]
  [ ( noarrow | arrow | reversearrow | twowayarrow ) ]
  [ linked [ item <number> ] <axis> [ using <expression> ] ]
```

The `set axis` command is used to add additional axes to plots and to configure their appearance. Where an axis is stated on its own, as in the example

```
set axis x2
```

additional horizontal or vertical axes are added with default settings. The converse statements

```
set noaxis x2
unset axis x2
```

are used, respectively, to remove axes from plots and to return them to their default configurations, which often has the same effect of removing them from the graph, unless they are configured otherwise in a configuration file.

The position of any axis can be explicitly set using syntax of the form:

```
set axis x top
set axis y right
set axis z back
```

Horizontal axes can be set to appear either at the `top` or `bottom`; vertical axes can be set to appear either at the `left` or `right`; and z-axes can be set to appear either at the `front` or `back`. By default, the `x1`-axis is placed along the bottom of graphs and the `y1`-axis is placed up the left-hand side of graphs. On three-dimensional plots, the `z1`-axis is placed at the front of the graph. The second set of axes are placed opposite to the first: the `x2`-, `y2`- and `z2`-axes are placed respectively along the top, right and back sides of graphs. Higher-numbered axes are placed alongside the `x1`-, `y1`- and `z1`-axes.

The following keywords may also be placed alongside the positional keywords listed above to specify how the axis should appear:

- `arrow` – Specifies that an arrowhead should be drawn on the right/top end of the axis. [**Not default**].
- `atzero` – Specifies that rather than being placed along an edge of the plot, the axis should mark the lines where the perpendicular axes `x1`, `y1` and/or `z1` are zero. [**Not default**].
- `automirrored` – Specifies that an automatic decision should be made between the behaviour of `mirrored` and `nomirrored`. If there are no axes on the opposite side of the graph, a mirror axis is produced. If there are already axes on the opposite side of the graph, no mirror axis is produced. [**Default**].
- `fullmirrored` – Similar to `mirrored`. Specifies that this axis should have a corresponding twin placed on the opposite side of the graph with mirroring ticks and labelling. [**Not default**; see `automirrored`].
- `invisible` – Specifies that the axis should not be drawn; data can still be plotted against it, but the axis is unseen. See Example 24 for a plot where all of the axes are invisible.
- `linked` – Specifies that the axis should be linked to another axis; see Section 8.8.9.
- `mirrored` – Specifies that this axis should have a corresponding twin placed on the opposite side of the graph with mirroring ticks but with no labels on the ticks. [**Not default**; see `automirrored`].
- `noarrow` – Specifies that no arrowheads should be drawn on the ends of the axis. [**Default**].
- `nomirrored` – Specifies that this axis should not have any corresponding twins. [**Not default**; see `automirrored`].
- `notatzero` – Opposite of `atzero`; the axis should be placed along an edge of the plot. [**Default**].
- `notlinked` – Specifies that the axis should no longer be linked to any other; see Section 8.8.9. [**Default**].
- `reversearrow` – Specifies that an arrowhead should be drawn on the left/bottom end of the axis. [**Not default**].
- `twowayarrow` – Specifies that arrowheads should be drawn on both ends of the axis. [**Not default**].
- `visible` – Specifies that the axis should be displayed; opposite of `invisible`. [**Default**].

11.52.5 axisunitstyle

```
set axisunitstyle ( bracketed | squarebracketed | ratio )
```

The setting `axisunitstyle` controls the style with which the units of plotted quantities are indicated on the axes of plots. The `bracketed` option causes the units to be placed in parentheses following the axis labels, whilst the `square-bracketed` option using square brackets instead. The `ratio` option causes the units to follow the label as a divisor so as to leave the quantity dimensionless.

11.52.6 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 setting `backup` 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 is turned off using the `set nobackup` command.

11.52.7 bar

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

The setting `bar` changes the size of the bar drawn on the end of the error bars, relative to the current point size. 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.

11.52.8 binorigin

```
set binorigin <value>
```

The setting `binorigin` affects the behaviour of the `histogram` command by adjusting where it places the boundaries between the bins into which it places data. The `histogram` command selects a system of bins which, if extended to infinity in both directions, would put a bin boundary at the value specified in the `set binorigin` command. Thus, if a value of 0.1 were specified to the `set binorigin` command, and a bin width of 20 were chosen by the `histogram` command, bin boundaries might lie at 20.1, 40.1, 60.1, and so on. The specified value may have any physical units, but must be real and finite.

11.52.9 binwidth

```
set binwidth <value>
```

The setting `binwidth` changes the width of the bins used by the `histogram` command. The specified width may have any physical units, but must be real and finite.

11.52.10 boxfrom

```
set boxfrom <value>
```

The setting `boxfrom` alters the vertical line from which bars are drawn when Pyxplot draws bar charts. By default, bars all originate from the line $y = 0$, but the example

```
set boxfrom 2
```

would make the bars originate from the line $y = 2$. The specified vertical abscissa value may have any physical units, but must be real and finite.

11.52.11 boxwidth

```
set boxwidth <width>
```

The setting `boxwidth` alters Pyxplot's behaviour when plotting bar charts. It sets the default width of the boxes used, in ordinate 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 positions. For example:

```
set boxwidth 2
```

would set all boxes to be two units wide, and

```
set boxwidth -2
```

would set all of the bars to have differing widths, centered upon their specified horizontal positions, such that their interfaces occur at the horizontal midpoints between them. The specified width may have any physical units, but must be real and finite.

11.52.12 c1format

```
set c1format ( auto | <format> )
      ( horizontal | vertical | rotate <angle> )
```

The `c1format` setting is used to manually specify an explicit format for the axis labels to take along the color scale bars drawn alongside plots which make use of the `colormap` plot style. It has similar syntax to the `set xformat` command.

11.52.13 c1label

```
set c1label <text> [ rotate <angle> ]
```

The setting `c1label` sets the label which should be written alongside the color scale bars drawn next to plots when the `colormap` plot style is used. An optional rotation angle may be specified to rotate axis labels clockwise by arbitrary angles. The angle should be specified either as a dimensionless number of degrees, or as a quantity with physical dimensions of angle.

11.52.14 calendar

```
set calendar [ ( input | output ) ] <calendar>
```

The `set calendar` command sets the calendar that Pyxplot uses to convert dates between calendar dates and Julian Day numbers. Pyxplot uses two separate calendars which may be different: an input calendar for processing dates that the user inputs as calendar dates, and an output calendar that controls how dates are displayed or written on plots. The available calendars are `British`, `French`, `Greek`, `Gregorian`, `Hebrew`, `Islamic`, `Jewish`, `Julian`, `Muslim`, `Papal` and `Russian`, where `Jewish` is an alias for `Hebrew` and `Muslim` is an alias for `Islamic`.

11.52.15 clip

```
set clip
```

The `set clip` command causes Pyxplot to clip points which extend over the edge of plots. The opposite effect is achieved using the `set noclip` command.

11.52.16 colorkey

```
set colorkey [ <position> ]
```

The setting `colorkey` determines whether color scales are drawn along the edges of plots drawn using the `colormap` plot style, indicating the mapping between represented values and colors. Note that such scales are only ever drawn when the `colormap` plot style is supplied with only three columns of data, since the color mappings are themselves multi-dimensional when more columns are supplied. Issuing the command

```
set colorkey
```

by itself causes such a scale to be drawn on graphs in the default position, usually along the right-hand edge of the graphs. The converse action is achieved by:

```
set nocolorkey
```

The command

```
unset colorkey
```

causes Pyxplot to revert to its default behaviour, as specified in a configuration file, if present. A position for the key may optionally be specified after the `set colorkey` command, as in the example:

```
set colorkey bottom
```

Recognised positions are `top`, `bottom`, `left` and `right`. `above` is an alias for `top`; `below` is an alias for `bottom` and `outside` is an alias for `right`.

11.52.17 colormap

```
set colormap <color expression> [ mask <expr> ]
```

The setting `colormap` is used to specify the mapping between ordinate values and colors used by the `colormap` plot style. Within the color expression, the variables `c1`, `c2`, `c3` and `c4` refer quantities calculated from the third through sixth columns of data supplied to the `colormap` plot style in a way determined by the `c<n>range` setting. Thus, the following color mapping, which is the default, produces a greyscale color mapping of the third column of data supplied to the `colormap` plot style; further columns of data, if supplied, are not used:

```
set c1range [*:*] renormalise
set colormap rgb(c1,c1,c1)
```

If a mask expression is supplied, then any areas in a color map where this expression evaluates to false (or zero) are made transparent.

11.52.18 contours

```
set contours [ ( <number> |
                 "(" { <value> } ")" ) ]
             [ (label | nolabel) ]
```

The setting `contours` is used to define the set of ordinate values for which contours are drawn when using the `contourmap` plot style. If `<number>` is specified, the contours are evenly spaced – either linearly or logarithmically, depending upon the state of the `logscale c1` setting – between the values specified in the `c1range` setting. Otherwise, the list of ordinate values may be specified as a ()-bracketed comma-separated list.

If the option `label` is specified, then each contour is labelled with the ordinate value that it follows. If the option `nolabel` is specified, then the contours are not labelled.

11.52.19 c<n>range

```
set c<n>range [ <range> ]
              [ reversed | noreversed ]
              [ renormalise | norenormalise ]
```

The `set c<n>range` command changes the range of ordinate values represented by different colors in the `colormap` plot style, and in the case of the `set c1range` command, also by contours in the `contourmap` plot style. The value `<n>` should be an integer in the range 1–4.

Contour Maps

The effect of the `set c1range` command upon the set of ordinate values for which contours are drawn in the `contourmap` plot style is dependent upon whether the `set contours` command has been supplied with a number of contours to draw, or a list of explicit ordinate values for which they should be drawn. In the latter case, the `set c1range` command has no effect. In the

former case, the contours are evenly spaced, either linearly or logarithmically depending upon the state of the `logscale c1` setting, between the minimum and maximum ordinate values supplied to the `set c1range` command. If an asterisk (`*`) is supplied in place of either the minimum and/or the maximum, then the range of values used is automatically scaled to fit the range of the data supplied.

Color Maps

The color of each pixel in a color map is determined by the `colormap` setting. This should contain an expression that evaluates to a color object, e.g. `rgb(c1,c2,c3)`, and which may take the variables `c1`, `c2`, `c3` and `c4` as parameters. The `colormap` plot style should be supplied with between three and six columns of data, the first two of which contain the x- and y-positions of points, and the remainder of which are used to set the values of the variables `c1`, `c2`, `c3` and `c4` when calculating the color with which that point should be represented. If fewer than six columns of data are supplied, then not all of these variables will be set.

The `set c<n>range` command is used to determine how the raw data values are mapped to the values of the variables `c1`–`c4`. If the `norenormalise` option is specified, then the raw values are passed directly to the expression. Otherwise, they are first scaled into the range zero to one. If an explicit range is specified to the `set c<n>range` command, then the upper limit of this range maps to the value one, and the lower limit maps to the value zero. This mapping is inverted if the `reverse` option is specified, such that the upper limit maps to zero, and the lower limit maps to one. If an asterisk (`*`) is supplied in place of either the upper and/or lower limit, then the range automatically scales to fit the data supplied. Intermediate values are scaled, either linearly or logarithmically, depending upon the state of the `logscale c<n>` setting. For more details of the syntax of the range specifier, see the `set xrange` command.

11.52.20 data style

See `set style data`.

11.52.21 display

```
set [no]display
```

By default, whenever an item is added to a multiplot canvas, or an existing item is moved or replotted, the whole multiplot is redrawn to reflect the change. This can be a time-consuming process when constructing large and complex multiplot canvases, as fresh output is produced at each step. For this reason, the `set nodisplay` command is provided, which stops Pyxplot from producing any graphical output. The `set display` command can subsequently be issued to return to normal behaviour. Scripts which produces large and complex multiplot canvases are typically wrapped as follows:

```
set nodisplay
...
set display
```

```
refresh
```

11.52.22 filter

```
set filter <filename wildcard> <filter command>
```

The `set filter` command allows input filter programs to be specified to allow Pyxplot to deal with file types that are not in the plaintext format which it ordinarily expects. Firstly the pattern used to recognise the filenames of the data files to which the filter should apply to must be specified; the standard wildcard characters `*` and `?` may be used. Then a filter program should be specified, along with any necessary command-line switches which should be passed to it. This program should take the name of the file to be filtered as the final option on its command line, immediately following any command-line switches specified above. It should output a suitable Pyxplot data fileon its standard output stream for Pyxplot to read. For example, to filter all files that end in `.foo` through the a program called `defoo` using the `--text` option one would type:

```
set filter "*.foo" "/usr/local/bin/defoo --text"
```

11.52.23 fontsize

```
set fontsize <value>
```

The setting `fontsize` changes the size of the font used to render all text labels which appear on graphs and multiplot canvases, including keys, axis labels, text labels produced using the `text` command, and so forth. The value specified should be a multiplicative factor greater than zero; a value of `2` would cause text to be rendered at twice its normal size, and a value of `0.5` would cause text to be rendered at half its normal size. The default value is one.

As an alternative, font sizes can be specified with coarser granulation directly in the latex text of labels, as in the example:

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

11.52.24 function style

See `set style function`.

11.52.25 grid

```
set [no]grid { <axis> }
```

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

```
set grid
```

would cause a grid to be drawn with its lines connecting to the ticks of the default axes (usually the first horizontal and vertical axes). Conversely, issuing the command

```
set nogrid
```

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 to draw gridlines from; in such cases, gridlines are then drawn only to connect with the ticks of the specified axes, as in the example

```
set grid x1 y3
```

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

```
set grid x1x2x3
```

11.52.26 gridmajcolor

```
set gridmajcolor <color>
```

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

```
set gridmajcolor purple
```

would cause the major gridlines to be drawn in purple. Any of the recognised color names listed in Section 19.4 can be used, or a numbered color from the present palette, or an object of type `color`.

See also the `set gridmincolor` command.

11.52.27 gridmincolor

```
set gridmincolor <color>
```

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

```
set gridmincolor purple
```

would cause the minor gridlines to be drawn in purple. Any of the recognised color names listed in Section 19.4 can be used, or a numbered color from the present palette, or an object of type `color`.

See also the `set gridmajcolor` command.

11.52.28 key

```
set key <position> [ <vector> ]
```

The setting `key` determines whether legends are drawn on graphs, and if so, where they should be located on the plots. Issuing the command

```
set key
```

by itself causes legends to be drawn on graphs in the default position, usually in the upper-right corner of the graphs. The converse action is achieved by:

```
set nokey
```

The command

```
unset key
```

causes Pyxplot to revert to its default behaviour, as specified in a configuration file, if present. A position for the key may optionally be specified after the `set key` command, as in the example:

```
set key bottom left
```

Recognised positions are `top`, `bottom`, `left`, `right`, `below`, `above`, `outside`, `xcenter` and `ycenter`. In addition, if none of these options quite achieve the desired position, a horizontal and vertical offset may be specified as a comma-separated pair after any of the positional keywords above. The first value is assumed to be the horizontal offset, and the second the vertical offset, both measured in centimeters. The example

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

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

11.52.29 keycolumns

```
set keycolumns ( <value> | auto )
```

The setting `keycolumns` sets how many columns the legend of a plot should be arranged into. By default, the legends of plots are arranged into an automatically-selected number of columns, equivalent to the behaviour achieved by issuing the command `set keycolumns auto`. However, if a different arrangement is desired, the `set keycolumns` command can be followed by any positive integer to specify that the legend should be split into that number of columns, as in the example:

```
set keycolumns 3
```

11.52.30 label

```
set label <label number> <text>
  [ <system> ] <x>, [ <system> ] <y>, [ [ <system> ] <z> ]
  [ rotate <angle> ]
  [ with { ( color <color> | fontsize <size> ) } ]
```

where `<system>` may take any of the values
`( first | second | screen | graph | axis<number> )`

The `set label` command is used to place text labels on graphs. The example

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

would place a label reading 'Hello' at the point $(0, 0)$ on a graph, as measured along the `x`- and `y`-axes. The tag `1` immediately following the keyword `label` is an identification number and allows the label to be subsequently removed using the `unset label` command. By default, the positional coordinates of the label are specified relative to the first horizontal and vertical axes, but they can alternatively be specified in any one of several coordinate systems. The coordinate system to be used is specified as in the example:

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

The name of the coordinate system to be used precedes the position value in that 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 centimeters from the origin of the graph. The syntax `axis<n>` may also be used, to use the n th horizontal or vertical 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
```

By default the labels are black; however, an arbitrary color may be specified using the `with color` modifier. For example,

```
set label 3 'A purple label' 0, 0 with color purple
```

would place a purple label at the origin.

11.52.31 linewidth

```
set linewidth <value>
```

The `set linewidth` command sets the default line width of the lines used to plot datasets onto graphs using plot styles such as `lines`, `errorbars`, etc. The value supplied should be a multiplicative factor relative to the default line width; a value of 1.0 would result in lines being drawn with their default thickness. For example, in the following statement, lines of three times the default thickness are drawn:

```
set linewidth 3
plot sin(x) with lines
```

The `set linewidth` command only affects plot statements where no line width is manually specified.

11.52.32 logscale

`set logscale { <axis> } [ <base> ]`

The setting `logscale` 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 horizontal and second vertical axes to be scaled logarithmically. Linear scaling can be restored to all axes using the command

```
set nolog
```

meanwhile the command

```
unset log
```

restores axes to their default scaling, as specified in any configuration file which may be present. Both of these commands can also be applied to only one or a selection of axes, as in the examples

```
set nolog x1 y2
```

and

```
unset log x1y2
```

Optionally, a base may be specified at the end of the `set logscale` command, to produce axes labelled in logarithms of arbitrary bases. The default base is 10.

In addition to acting upon any combination of x-, y- and z-axes, the `set logscale` command may also be requested to set the `c1`, `c2`, `c3`, `c4`, `t`, `u` and/or `v` axes to scale logarithmically. The first four of these options affect whether the colors on color maps scale linearly or logarithmically with input ordinate values; see the `set c<n>range` command for more details. The final three of these options specifies whether parametric functions are sampled linearly or logarithmically in the variables `t` (one-dimensional), or `u` and `v` (two-dimensional); see the `set trange`, `set urange` and `set vrange` commands for more details.

11.52.33 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 10.2 for a full discussion of multiplot mode.

11.52.34 mxtics

See `set xtics`.

11.52.35 mytics

See `set xtics`.

11.52.36 mztics

See `set ztics`.

11.52.37 noarrow

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

Issuing the command

```
set noarrow
```

removes all arrows configured with the `set arrow` command. Alternatively, individual arrows can be removed using commands of the form

```
set noarrow 2
```

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

11.52.38 noaxis

```
set noaxis [ { <axis> } ]
```

The `set noaxis` command is used to remove axes from graphs; it achieves the opposite effect from the `set axis` command. It should be followed by a comma-separated lists of the axes which are to be removed from the current axis configuration.

11.52.39 nobackup

See `backup`.

11.52.40 noclip

See `clip`.

11.52.41 nocolorkey

```
set nocolorkey
```

Issuing the command `set nocolorkey` causes plots to be generated with no color scale when the `colormap` plot style is used. See the `set colorkey` command for more details.

11.52.42 nodisplay

See `display`.

11.52.43 nogrid

```
set nogrid { <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 be selectively removed. The following example would remove gridlines associated with the first horizontal axis and the second vertical axis:

```
set nogrid x1 y2
```

11.52.44 nokey

```
set nokey
```

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

11.52.45 nolabel

```
set nolabel { <label number> }
```

Issuing the command

```
set nolabel
```

removes all text labels configured using the `set label` command. Alternatively, individual labels can be removed using the syntax:

```
set nolabel 2
```

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

11.52.46 nologscale

```
set nologscale { <axis> }
```

The setting `nologscale` causes an axis to be laid out with linearly, rather than logarithmically, spaced intervals; it is equivalent to the setting `linearscale`. It is the converse of the setting `logscale`. 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 horizontal and second vertical axes to be scaled linearly.

11.52.47 nomultiplot

```
set nomultiplot
```

The `set nomultiplot` command causes Pyxplot to leave multiplot mode; outside of multiplot mode, only single graphs and vector graphics objects are displayed at any one time, whereas in multiplot mode, galleries of plots and vector graphics can be placed alongside one another. See Section 10.2 for a full discussion of multiplot mode.

11.52.48 nostyle

```
set nostyle <style number>
```

The setting `nostyle` deletes a numbered plot style set using the `set style` command. For example, the command

```
set nostyle 3
```

would delete the third numbered plot style, if defined. See the command `set style` for more details.

11.52.49 notitle

```
set notitle
```

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

11.52.50 noxtics

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

This command causes graphs to be produced with no major tick marks along the specified axis. For example, the `set noxtics` command removes all major tick marks from the x-axis.

11.52.51 noytics

Similar to the `set noxtics` command, but acts on vertical axes.

11.52.52 noztics

Similar to the `set noxtics` command, but acts on z-axes.

11.52.53 numerics

```
set numerics [ ( complex | real ) ] [ errors ( explicit | quiet) ]
    [ display ( latex | natural | typeable) ]
    [ sigfig <precision> ]
```

The `set numerics` command is used to adjust the way in which calculations are carried out and numerical quantities are displayed:

- The option `complex` causes Pyxplot to switch from performing real arithmetic (default) to performing complex arithmetic. The option `real` causes any calculations which return results with finite imaginary components to generate errors.
- The option `errors` controls how numerical errors such as divisions by zero, numerical overflows, and the querying functions outside of the domains in which they are defined, are communicated to the user. The option `explicit` (default) causes an error message to be displayed on the terminal whenever a calculation causes an error. The option `quiet` causes such calculations to silently generate a `nan` (not a number) result. The latter is especially useful when, for example, plotting an expression with the ordinate axis range set to extend outside the domain in which that expression returns a well-defined real result; it suppresses the error messages which might otherwise result from Pyxplot's attempts to evaluate the expression in those domains where its result is undefined. The option `nan` is a synonym for `quiet`.
- The setting `display` changes the format in which numbers are displayed on the terminal. Setting the option to `typeable` causes the numbers to be printed in a form suitable for pasting back into Pyxplot commands. The setting `latex` causes latex-compatible output to be generated. The setting `natural` generates concise, human-readable output which has neither of the above properties.
- The setting `sigfig` changes the number of significant figures to which numbers are displayed on the Pyxplot terminal. Regardless of the value set, all calculations are internally carried out and stored at double precision, accurate to around 16 significant figures.

11.52.54 origin

```
set origin <vector>
```

The `set origin` command is used to set the location of the bottom-left corner of the next graph to be placed on a multiplot canvas. For example, the command

```
set origin 3,5
```

would cause the next graph to be plotted with its bottom-left corner at position $(3, 5)$ centimeters on the multiplot canvas. Alternatively, either of the coordinates may be specified as quantities with physical units of length, such as `unit(35*mm)`. The `set origin` command is of little use outside of multiplot mode.

11.52.55 output

```
set output <filename>
```

The setting `output` 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`.

11.52.56 palette

```
set palette { <color> }
```

Pyxplot has a palette of colors which it assigns sequentially to datasets when colors are not manually assigned. This is also the palette to which reference is made if the user issues a command such as

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

requesting the fifth color from the palette. By default, this palette contains a range of distinctive colors. However, the user can choose to substitute his own list of colors using the `set palette` command. It should be followed by a comma-separated list of color 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 color names listed in Section 19.4 can be used, or a numbered color from the present palette, or an object of type `color`.

11.52.57 papersize

```
set papersize ( <named size> | <height>,<width> )
```

The setting `papersize` changes the size of output produced by the `postscript` terminal, and whenever the `enlarge` terminal option is set (see the `set terminal` command). This can take the form of either a recognised paper size name – a list of these is given in Appendix 16 – or as a (height, width) pair of values, both measured in millimeters. The following examples demonstrate these possibilities:

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

11.52.58 pointlinewidth

```
set pointlinewidth <value>
```

The setting `pointlinewidth` changes the width of the lines that are used to plot data points. For example,

```
set pointlinewidth 20
```

would cause points to be plotted with lines 20 times the default thickness. The setting `pointlinewidth` can be abbreviated as `plw`.

11.52.59 pointsize

```
set pointsize <value>
```

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

```
set pointsize 1.5
```

would cause points to be drawn at 1.5 times their default size.

11.52.60 preamble

```
set preamble <text>
```

The setting `preamble` changes the text of the preamble that is passed to latex prior to the rendering of each text item on the current multiplot canvas. This allows, for example, different packages to be loaded by default and user-defined macros to be set up, as in the examples:

```
set preamble \usepackage{marvosym}
set preamble \def\degrees{$^\circ$}
```

11.52.61 samples

```
set samples [<value>]
            [grid <x_samples> [x] <y_samples>]
            [interpolate ( inverseSquare |
                           monaghanLattanzio |
                           nearestNeighbor ) ]
```

The setting `samples` determines the number of values along the ordinate axis at which functions are evaluated when they are plotted. For example, the command

```
set samples 100
```

would cause functions to be evaluated at 100 points along the ordinate axis. Increasing this value will cause functions to be plotted more smoothly, but also more slowly, and the PostScript files generated will also be larger. When functions are plotted with the `points` plot style, this setting controls the number of points plotted.

After the keyword `grid` may be specified the dimensions of the two-dimensional grid of samples used in the `colormap` and `surface` plot styles, and internally when calculating the contours to be plotted in the `contourmap` plot style. If a `*` is given in place of either of the dimensions, then the same number of samples as are specified in `<value>` are taken.

After the keyword `interpolate`, the method used for interpolating non-gridded two-dimensional data onto the above-mentioned grid may be specified. The available options are `InverseSquare`, `MonaghanLattanzio` and `Nearest-Neighbour`.

11.52.62 seed

```
set seed <value>
```

The `set seed` command sets the seed used by all of those mathematical functions which generate random samples from probability distributions. This allows repeatable sequences of pseudo-random numbers to be generated. Upon initialisation, Pyxplot returns the sequence of random numbers obtained after issuing the command `set seed 0`.

11.52.63 size

```
set size [<width>]
         [( ratio <ratio> | noratio  | square)]
         [(zratio <ratio> | nozratio )]
```

The setting `size` is used to set the width or aspect ratio of the next graph to be generated. If a width is specified, then it may either take the form of a dimensionless number implicitly measured in centimeters, or a quantity with physical dimensions of length such as `unit(50*mm)`.

When the keyword `ratio` is specified, it should be followed by the ratio of the graph's height to its width, i.e. of the length of its y-axes to that of its x-axes. The keyword `noratio` returns the aspect ratio to its default value of the golden ratio, and the keyword `square` sets the aspect ratio to one.

When the keyword `zratio` is specified, it should be followed by the ratio of the length of three-dimensional graphs' z-axes to that of their x-axes. The keyword `nozratio` returns this aspect ratio to its default value of the golden ratio.

noratio

```
set size noratio
```

Executing the command

```
set size noratio
```

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

ratio

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

This command 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, multiplied by this aspect ratio. 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 $2/(1+\sqrt{5})$.

square

```
set size square
```

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

11.52.64 style

```
set style <style number> {<style option>}
```

At times, the string of style keywords following the `with` modifier in plot commands can grow rather unwieldy in its length. For clarity, frequently used plot styles can be stored as numbered plot `styles`. The syntax for setting a numbered plot style is:

```
set style 2 points pointtype 3
```

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

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

11.52.65 style data — style function

```
set style { data | function } {<style option>}
```

The `set style data` command affects the default style with which data from files is plotted. Likewise the `set style function` command changes the default style with which functions are plotted. Any valid style modifier which can follow the keyword `with` in the `plot` command can be used. For example, the commands

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

would cause data files to be plotted, by default, using points and functions using lines with the first defined line style.

11.52.66 terminal

```
set terminal ( X11_singleWindow | X11_multiWindow | X11_persist |
               bmp | eps | gif | jpeg | pdf | png | postscript |
               svg | tiff )
             ( color | color | monochrome )
             ( dpi <value> )
             ( portrait | landscape )
             ( invert | noinvert )
             ( transparent | solid )
             ( antialias | noantialias )
             ( enlarge | noenlarge )
```

The `set terminal` command controls the graphical format in which Pyxplot renders plots and multiplot canvases, for example configuring 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 graphical formats which Pyxplot supports using this command.

The following graphical formats are supported: `X11_singleWindow`, `X11_multiWindow`, `X11_persist`, `bmp`, `eps`, `gif`, `jpeg`, `pdf`, `png`, `postscript`, `svg`[2], `tiff`. 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: `color`, `monochrome`, `dpi`, `portrait`, `landscape`, `invert`, `noinvert`, `transparent`, `solid`, `enlarge`, `noenlarge`. Details of each of these can be found below.

antialias

The `antialias` terminal option causes plots produced with the bitmap terminals (i.e. `bmp`, `gif`, `jpeg`, `png` and `tiff`) to be antialiased; this is the default behaviour.

bmp

The `bmp` terminal renders output as Windows bitmap images. The filename to which output is to be sent should be set using the `set output` command; the default is `pyxplot.bmp`. The number of dots per inch used can be changed using the `dpi` option. The `invert` option may be used to produce an image with inverted colors.

color

The `color` terminal option causes plots to be produced in color; this is the default behaviour.

color

The `color` terminal option is the US-English equivalent of `color`.

dpi

When Pyxplot is set to produce bitmap graphics output, using the `bmp`, `gif`, `jpg` or `png` terminals, the setting `dpi` changes the number of dots per inch with which these graphical images are produced. That is to say, it changes the image resolution of the output images. For example,

```
set terminal dpi 100
```

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

[2]The `svg` output terminal is experimental and may be unstable. It relies upon the use of the `svg` output device in Ghostscript, which may not be present on all systems.

enlarge

The `enlarge` terminal option causes plots and multiplot canvases to be enlarged or shrunk to fit within the margins of the currently selected paper size. It is especially useful when using the `postscript` terminal, as it allows for the production of immediately-printable output.

eps

Sends output to Encapsulated PostScript (`eps`) files. The filename to which output should be sent can be set using the `set output` command; the default is `pyxplot.eps`. This terminal produces images suitable for including in, for example, latex documents.

gif

The `gif` terminal renders output as gif images. The filename to which output should be sent can 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. Transparent gifs can be produced with the `transparent` option. The `invert` option may be used to produce an image with inverted colors.

invert

The `invert` terminal option causes the bitmap terminals (i.e. `bmp`, `gif`, `jpeg`, `png` and `tiff`) to produce output with inverted colors.

jpeg

The `jpeg` terminal renders output as jpeg images. The filename to which output should be sent can 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. The `invert` option may be used to produce an image with inverted colors.

landscape

The `landscape` terminal option causes Pyxplot's output to be displayed in rotated orientation. This can be useful for fitting graphs onto sheets of paper, but is generally less useful for plotting things on screen.

monochrome

The `monochrome` terminal option causes plots to be rendered in black and white. This changes the default behaviour of the `plot` command to be optimised for monochrome display, and so, for example, different dash styles are used to differentiate between lines on plots with several datasets.

noantialias

The `noantialias` terminal option causes plots produced with the bitmap terminals (i.e. `bmp`, `gif`, `jpeg`, `png` and `tiff`) not to be antialiased. This can be useful when making plots which will subsequently have regions cut out and made transparent.

noenlarge

The `noenlarge` terminal option causes the output not to be scaled to fit within the margins of the currently-selected papersize. This is the opposite of `enlarge` option.

noinvert

The `noinvert` terminal option causes the bitmap terminals (i.e. `gif`, `jpeg`, `png`) to produce normal output without inverted colors. This is the opposite of the `inverse` option.

pdf

The `pdf` terminal renders output in Adobe's Portable Document Format (PDF).

png

The `png` terminal renders output as png images. The filename to which output should be sent can 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. Transparent pngs can be produced with the `transparent` option. The `invert` option may be used to produce an image with inverted colors.

portrait

The `portrait` terminal option causes Pyxplot's output to be displayed in upright (normal) orientation; it is the converse of the `landscape` option.

postscript

The `postscript` terminal renders output as PostScript files. The filename to which output should be sent can 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; it should not be used for producing images to be embedded in documents, for which the `eps` terminal should be used.

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

The `X11_multiwindow` terminal displays plots on the screen in X11 windows. 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. When Pyxplot exits, however, all of the windows are closed.

X11_persist

The `X11_persist` terminal displays plots on the screen in X11 windows. 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

The `X11_singlewindow` terminal displays plots on the screen in X11 windows. 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.

11.52.67 textcolor

```
set textcolor <color>
```

The setting `textcolor` changes the default color of all text displayed on plots or multiplot canvases. For example,

```
set textcolor red
```

causes all text labels, including the labels on graph axes and legends, etc. to be rendered in red. Any of the recognised color names listed in Section 19.4 can be used, or a numbered color from the present palette, or an object of type `color`.

11.52.68 texthalign

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

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

```
set texthalign left
set texthalign center
set texthalign right
```

11.52.69 textvalign

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

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

```
set textvalign bottom
set textvalign center
set textvalign top
```

11.52.70 timezone

```
set timezone <timezone>
```

The `set timezone` command sets the name of the default timezone that Pyxplot uses when handling date objects. It should take the form of a `tz database` timezone name, for example `Europe/London`. A complete list of these can be found here: http://en.wikipedia.org/wiki/List_of_tz_database_time_zones. If no timezone is specified, then the default set using the `set timezone` command is used. If universal time is required, `UTC` may be specified as the timezone. For example:

```
set timezone Europe/Paris
set timezone Australia/Perth
set timezone America/New_York
set timezone Antarctica/South_Pole
set timezone UTC
```

Note that it is *not* permitted to set a timezone such as `GMT`, `EDT` or `CEST`; a place should be specified, and Pyxplot will use the local time from that location.

11.52.71 title

```
set title <title>
```

The setting `title` 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 using the command:

```
set notitle
```

11.52.72 trange

```
set trange [ <range> ] [reverse]
```

The `set trange` command changes the range of the free parameter `t` used when generating parametric plots. For more details of the syntax of the range specifier, see the `set xrange` command. Note that `t` is not allowed to autoscale, and so the `*` character is not permitted in the specified range.

11.52.73 unit

```
set unit [ angle ( dimensionless | nodimensionless ) ]
         [ of <dimension> <unit> ]
         [ scheme <unit scheme> ]
         [ preferred <unit> ]
         [ nopreferred <unit> ]
         [ display ( full | abbreviated | prefix | noprefix ) ]
```

The `set unit` command controls how quantities with physical units are displayed by Pyxplot. The `set unit scheme` command provides the most general configuration option, allowing one of several *units schemes* to be selected, each of which comprises a list of units which are deemed to be members of that particular scheme. For example, in the CGS unit scheme, all lengths are displayed in centimeters, all masses are displayed in grammes, all energies are displayed in ergs, and so forth. In the imperial unit scheme, quantities are displayed in British imperial units – inches, pounds, pints, and so forth – and in the US unit scheme, US customary units are used. The available schemes are: `ancient`, `cgs`, `imperial`, `planck`, `si`, and `us`.

To fine-tune the unit used to display quantities with a particular set of physical dimensions, the `set unit of` form of the command should be used. For example, the following command would cause all lengths to be displayed in inches:

```
set unit of length inch
```

The `set unit preferred` command offers a slightly more flexible way of achieving the same result. Whereas the `set unit of` command can only operate on named quantities such as lengths and powers, and cannot act upon compound units such as `W/Hz`, the `set unit preferred` command can act upon any unit or combination of units, as in the examples:

```
set unit preferred parsec
set unit preferred W/Hz
set unit preferred N*m
```

The latter two examples are particularly useful when working with spectral densities (powers per unit frequency) or torques (forces multiplied by distances). Unfortunately, both of these units are dimensionally equal to energies, and so are displayed by Pyxplot in Joules by default. The above statement overrides such behaviour. Having set a particular unit to be preferred, this can be unset as in the following example:

```
set unit nopreferred parsec
```

By default, units are displayed in their abbreviated forms, for example `A` instead of `amperes` and `W` instead of `watts`. Furthermore, SI prefixes such as milli- and kilo- are applied to SI units where they are appropriate. Both of these behaviours can be turned on or off, in the former case with the commands

```
set unit display abbreviated
set unit display full
```

and in the latter case using the following pair of commands:

```
set unit display prefix
set unit display noprefix
```

11.52.74 urange

```
set urange [ <range> ] [reverse]
```

The `set urange` command changes the range of the free parameter `u` used when generating parametric plots sampled over grids of (`u`,`v`) values. For more details of the syntax of the range specifier, see the `set xrange` command. Note that `u` is not allowed to autoscale, and so the `*` character is not permitted in the specified range.

Specifying the `set urange` command by itself specified that parametric plots should be sampled over two-dimensional grids of (`u`,`v`) values, rather than one-dimensional ranges of `t` values.

11.52.75 view

```
set view <theta>, <phi>
```

The `set view` command is used to specify the angle from which three-dimensional plots are viewed. It should be followed by two angles, which can either be expressed in degrees, as dimensionless numbers, or as quantities with physical units of angle:

```
set view 60,30

set unit angle nodimensionless
set view unit(0.1*rev),unit(2*rad)
```

The orientation $(0, 0)$ corresponds to having the x-axis horizontal, the z-axis vertical, and the y-axis directed into the page. The first angle supplied to the `set view` command rotates the plot in the (x, y) plane, and the second angle tips the plot up in the plane containing the z-axis and the normal to the user's two-dimensional display.

11.52.76 viewer

```
set viewer ( auto | <command> )
```

The `set viewer` command is used to select which external PostScript viewing application is used to display Pyxplot output on screen in the `X11` terminals. If the option `auto` is selected, then either ghostview or `ggv` is used, if installed. Alternatively, any other application such as `evince` or `okular` can be selected by name, providing it is installed in within your shell's search path or an absolute path is provided, as in the examples:

```
set viewer evince
set viewer /usr/bin/okular
```

Additional commandline switches may also be provided after the name of the application to be used, as in the example

```
set viewer gv --grayscale
```

11.52.77 vrange

```
set vrange [ <range> ] [reverse]
```

See the `set urange` command.

11.52.78 width

```
set width <value>
```

The setting `width` is used to set the width of the next graph to be generated. The width is specified either as a dimensionless number implicitly measured in centimeters, or as a quantity with physical dimensions of length such as `unit(50*mm)`.

11.52.79 xformat

```
set <axis>format ( auto | <format> )
      ( horizontal | vertical | rotate <angle> )
```

By default, the major tick marks along axes are labelled with representations of the ordinate values at each point, each accurate to the number of significant figures specified using the `set numerics sigfig` command. These labels may appear as decimals, such as 3.142, in scientific notion, as in 3×10^8, or, on logarithmic axes where a base has been specified for the logarithms, using syntax such as[3]

```
set log x1 2
```

in a format such as 1.5×2^8.

The `set xformat` command – together with its companions such as `set yformat` – is used to manually specify an explicit format for the axis labels to take, as demonstrated by the following pair of examples:

```
set xformat "%.2f"%(x)
set yformat "%s$^\prime$"%(y/unit(feet))
```

The first example specifies that values should be displayed to two decimal places along the `x`-axis; the second specifies that distances should be displayed in feet along the `y`-axis. Note that the dummy variable used to represent the represented value is `x`, `y` or `z` depending upon the direction of the axis, but that the dummy variable used in the `set x2format` command is still `x`. The following pair of examples both have the equivalent effect of returning the `x2`-axis to its default system of tick labels:

```
set x2format auto
set x2format "%s"%(x)
```

The following example specifies that ordinate values should be displayed as multiples of π:

[3] Note that the `x` axis must be referred to as `x1` here to distinguish this statement from `set log x2`.

```
set xformat "%s$\pi$"%(x/pi)
plot [-pi:2*pi] sin(x)
```

Note that where possible, Pyxplot intelligently changes the positions along axes where it places the ticks to reflect significant points in the chosen labelling system. The extent to which this is possible depends upon the format string supplied. It is generally easier when continuous-varying numerical values are substituted into strings, rather than discretely-varying values or strings.

11.52.80 xlabel

```
set <axis>label <text> [ rotate <angle> ]
```

The setting `xlabel` sets the label which should be written along the `x`-axis. For example,

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

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

```
set x10label 'foo'
```

would label the tenth horizontal axis. Similarly, labels can be placed on vertical axes as follows:

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

An optional rotation angle may be specified to rotate axis labels clockwise by arbitrary angles. The angle should be specified either as a dimensionless number of degrees, or as a quantity with physical dimensions of angle.

11.52.81 xrange

```
set <axis>range <range> [reverse]
```

The setting `xrange` controls the range of values spanned by 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 horizontal axis to run from 0 to 10. Higher numbered axes may be referred to be inserting their number after the `x`; the ranges of vertical axes may similarly be set by replacing the `x` with a `y`. Hence,

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

sets the range of the 23rd vertical axis to run from -5 to 5. To request a range to be automatically scaled an asterisk can be used. The following command would set the `x`-axis to have an upper limit of 10, but does not affect the lower limit; its range remains at its previous setting:

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

The keyword `reverse` is used to indicate that the two limits of an axis should be swapped. This is useful for setting auto-scaling axes to be displayed running from right to left, or from top to bottom.

11.52.82 xtics

```
set [m]<axis>tics
    [ ( axis | border | inward | outward | both ) ]
    [ ( autofreq
          | [<minimum>,] <increment> [, <maximum>]
          | "(" { [ <label> ] <position> } ")"
        ] )
```

By default, Pyxplot places a series of tick marks at significant points along each axis, with the most significant points being labelled. Labelled tick marks are termed *major* ticks, and unlabelled tick marks are termed *minor* ticks. The position and appearance of the major ticks along the `x`-axis can be configured using the `set xtics` command; the corresponding `set mxtics` command configures the appearance of the minor ticks along the `x`-axis. Analogous commands such as `set ytics` and `set mx2tics` configure the major and minor ticks along other axes.

The keywords `inward`, `outward` and `both` are used to configure how the ticks appear – whether they point inward, towards the plot, as is default, or outwards towards the axis labels, or in both directions. The keyword `axis` is an alias for `inward`, and `border` is an alias for `outward`.

The remaining options are used to configure where along the axis ticks are placed. If a series of comma-separated values `<minimum>, <increment>, <maximum>` are specified, then ticks are placed at evenly spaced intervals between the specified limits. The `<minimum>` and `<maximum>` values are optional; if only one value is specified then it is taken to be the step size between ticks. If two values are specified, then the first is taken to be `<minimum>`. In the case of logarithmic axes, `<increment>` is applied as a multiplicative step size.

Alternatively, if a bracketed list of quoted tick labels and tick positions are provided, then ticks can be placed on an axis manually and each given its own textual label. The quoted tick labels may be omitted, in which case they are automatically generated:

```
set xtics ("a" 1, "b" 2, "c" 3)
set xtics (1,2,3)
```

The keyword `autofreq` overrides any manual selection of ticks which may have been placed on an axis and resumes the automatic placement of ticks along it. The `show xtics` command, together with its companions such as `show x2tics` and `show ytics`, is used to query the current ticking options. The `set noxtics` command is used to stipulate that no ticks should appear along a particular axis:

```
set noxtics
show xtics
```

11.52.83 yformat

See `xformat`.

11.52.84 ylabel

See `xlabel`.

11.52.85 yrange

See `xrange`.

11.52.86 ytics

See `xtics`.

11.52.87 zformat

See `xformat`.

11.52.88 zlabel

See `xlabel`.

11.52.89 zrange

See `xrange`.

11.52.90 ztics

See `xtics`.

11.53 show

```
show { all | axes | functions | settings | units
       | userfunctions | variables | <parameter> }
```

The `show` command displays the present state of parameters which can be set with the `set` command. For example,

```
show pointsize
```

displays the currently set point size.

Details of the various parameters which can be queried 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 a complete list of the present values of all of Pyxplot's configurable parameters. The command `show settings` shows all of these parameters, but does not list the currently-configured variables, functions and axes. `show axes` shows the configuration states of all graph axes. `show variables` lists all of the currently defined variables. And finally, `show functions` lists all of the current user-defined functions.

11.54 solve

```
solve { <equation> } via { <variable> }
```

The `solve` command can be used to solve simple systems of one or more equations numerically. It takes as its arguments a comma-separated list of the equations which are to be solved, and a comma-separated list of the variables which are to be found. The latter should be prefixed by the word `via`, to separate it from the list of equations.

Note that the time taken by the solver dramatically increases with the number of variables which are simultaneously found, whereas the accuracy achieved simultaneously decreases. The following example solves a simple pair of simultaneous equations of two variables:

```
pyxplot> solve x+y=10, x-y=3 via x,y
pyxplot> print x
6.5
pyxplot> print y
3.5
```

No output is returned to the terminal if the numerical solver succeeds, otherwise an error message is displayed. If any of the fitting variables are already defined prior to the `solve` command's being called, their values are used as initial guesses, otherwise an initial guess of unity for each fitting variable is assumed. Thus, the same `solve` command returns two different values in the following two cases:

```
pyxplot> x= # Undefine x
pyxplot> solve cos(x)=0 via x
pyxplot> print x/pi
0.5
pyxplot> x=10
pyxplot> solve cos(x)=0 via x
pyxplot> print x/pi
3.5
```

In cases where any of the variables being solved for are not dimensionless, it is essential that an initial guess with appropriate units be supplied, otherwise the solver will try and fail to solve the system of equations using dimensionless values:

```
x = unit(m)
y = 5*unit(km)
solve x=y via x
```

The `solve` command works by minimising the squares of the residuals of all of the equations supplied, and so even when no exact solution can be found, the best compromise is returned. The following example has no solution – a system of three equations with two variables is over-constrained – but Pyxplot nonetheless finds a compromise solution:

```
pyxplot> solve x+y=10, x-y=3, 2*x+y=16 via x,y
pyxplot> print x
6.4220634
pyxplot> print y
3.4266948
```

When complex arithmetic is enabled, the `solve` command allows each of the variables being fitted to take any value in the complex plane, and thus the number of dimensions of the fitting problem is effectively doubled – the real and imaginary components of each variable are solved for separately – as in the following example:

```
pyxplot> set numerics complex
pyxplot> solve exp(x)=e*i via x
pyxplot> print Re(x)
-1227.7
pyxplot> print Im(x)/pi
0
```

11.55 spline

```
spline [ <range> ] <function name>()
       ( <filename> | { <expression> } | { <vector obj> } )
       [ every { <expression> } ]
       [ index <value> ]
       [ select <expression> ]
       [ using { <expression> } ]
```

The `spline` command is an alias for the `interpolate spline` command. See the entry for the `interpolate` command for more details.

11.56 swap

```
swap <item1> <item2>
```

Items on multiplot canvases are drawn in order of increasing identification number, and thus items with low identification numbers are drawn first, at the back of the multiplot, and items with higher identification numbers are later, towards the front of the multiplot. When new items are added, they are given higher identification numbers than previous items and appear at the front of the multiplot.

If this is not the desired ordering, then the `swap` command may be used to rearrange items. It takes the identification numbers of two multiplot items and swaps their identification numbers and hence their positions in the ordered sequence. Thus, if, for example, the corner of item 3 disappears behind the corner of item 5, when the converse effect is actually desired, the following command should be issued:

```
swap 3 5
```

11.57 tabulate

```
tabulate [ <range> ]
       ( <filename> | { <expression> } | { <vector obj> } )
```

```
[ every { <expression> } ]
[ index <value> ]
[ select <expression> ]
[ sortby <expression> ]
[ using { <expression> } ]
[ with <output format> ]
```

Pyxplot's `tabulate` command is similar to its `plot` command, but instead of plotting a series of data points onto a graph, it outputs them to data files. This can be used to produce text files containing samples of functions, to rearrange/filter the columns in data files, to change the units in which data is expressed in data files, and so forth. The following example would produce a data file called `gamma.dat` containing a list of values of the gamma function:

```
set output 'gamma.dat'
tabulate [1:5] gamma(x)
```

Multiple functions may be tabulated into the same file, either by using the `using` modifier:

```
tabulate [0:2*pi] sin(x):cos(x):tan(x) u 1:2:3:4
```

or by placing them in a comma-separated list, as in the `plot` command:

```
tabulate [0:2*pi] sin(x), cos(x), tan(x)
```

In the former case, the functions are tabulated horizontally alongside one another in a series of columns. In the latter case, the functions are tabulated one after another in a series of index blocks separated by double linefeeds (see Section 3.8).

The setting `samples` can be used to control the number of points that are produced when tabulating functions, in the same way that it controls the `plot` command:

```
set samples 200
```

If the ordinate axis is set to be logarithmic then the points at which functions are evaluated are spaced logarithmically, otherwise they are spaced linearly.

The `select`, `using` and `every` modifiers operate in the same manner in the `tabulate` command as in the `plot` command. Thus, the following example would write out the third, sixth and ninth columns of the data file `input.dat`, but only when the arcsine of the value in the fourth column is positive:

```
set output 'filtered.dat'
tabulate 'input.dat' u 3:6:9 select (asin($4)>0)
```

The numerical display format used in each column of the output file is chosen automatically to preserve accuracy whilst simultaneously being as easily human readable as possible. Thus, columns which contain only integers are displayed as such, and scientific notation is only used in columns which contain very large or very small values. If desired, however, a format statement may be specified using the `with format` specifier. The syntax for this is similar to that expected by the string substitution operator (`%`; see Section 6.2.1). As an example, to tabulate the values of x^2 to very many significant figures with some additional text, one could use:

```
tabulate x**2 with format "x = %f ; x**2 = %27.20e"
```

This might produce the following output:

```
x = 0.000000 ; x**2 =  0.00000000000000000000e+00
x = 0.833333 ; x**2 =  6.94444444444442421371e-01
x = 1.666667 ; x**2 =  2.77777777777778167589e+00
```

The data produced by the `tabulate` command can be sorted in order of any arbitrary metric by supplying an expression after the `sortby` modifier; where such expressions are supplied, the data is sorted in order from the smallest value of the expression to the largest.

11.58 text

```
text [ item <id> ] <text string> [ at <vector> ]
     [ rotate <angle> ] [ gap <length> ]
     [ halign <alignment> ] [ valign <alignment> ]
     [ with color <color> ]
```

The `text` command allows strings of text to a added as labels on multiplot canvases. The example

```
text 'Hello World!' at 0,2
```

would render the text 'Hello World!' at position $(0, 2)$, measured in centimeters. The alignment of the text item with respect to this position can be set using the `set texthalign` and `set textvalign` commands, or using the `halign` and `valign` modifiers supplied to the `text` command itself.

A gap may be specified, which should either have dimensions of length, or be dimensionless, in which case it is interpreted as being measured in centimeters. If a gap is specified, then the text string is slightly displaced from the specified position, in the direction in which it is being aligned.

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
```

By default the text is black; however, an arbitrary color may be specified using the `with color` modifier. For example:

```
text 'A purple label' at 0, 0 with color purple
```

would add a purple label at the origin of the multiplot.

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.

All vector graphics objects placed on multiplot canvases receive unique identification numbers which count sequentially from one, and which may be listed using the `list` command. By reference to these numbers, they can be deleted and subsequently restored with the `delete` and `undelete` commands respectively.

11.59 undelete

```
undelete { <item number> }
```

The `undelete` command allows vector graphics objects which have previously been deleted from the current multiplot canvas to be restored. The item(s) which are to be restored should be identified using the reference number(s) which were used to delete them, and can be queried using the `list` command. The example

```
undelete 1
```

would cause the previously deleted item numbered `1` to reappear.

11.60 unset

```
unset <setting>
```

The `unset` command causes a configuration option which 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.

Any keyword which can follow the `set` command to identify a configuration parameter can also follow the `unset` command; a complete list of these can be found under the `set` command.

11.61 while

```
while <condition> [ loopname <loopname> ]
  <code>
```

The `while` command executes a block of commands repeatedly, checking the provided condition at the start of each iteration. If the condition is true, the loop executes again. This is similar to a `do` loop, except that the contents of a `while` loop may not be executed at all if the iteration criterion tests false upon the first iteration. For example, the following code prints out the low-valued Fibonacci numbers:

```
i = 1
j = 1
while (j < 50)
 {
  print j
  i = i + j
  print i
  j = j + i
 }
```

Chapter 12

List of in-built functions

The following is a complete list of the default functions which are built into Pyxplot. Except where stated otherwise, functions may be assumed to expect numerical arguments. Where arguments are represented by the letter x, they must usually be real numbers. Where arguments are represented by the letter z, they are usually permitted to be complex numbers. Functions which are defined within Pyxplot's default modules, but which are not in its default namespace, are listed in subsections below.

abs(z)
The abs(z) function returns the absolute magnitude of z, where z may be any general complex number. The output shares the physical dimensions of z, if any.

acos(z)
The acos(z) function returns the arccosine of z, where z may be any general dimensionless complex number. The output has physical dimensions of angle.

acosec(z)
The acosec(z) function returns the arccosecant of z, where z may be any general dimensionless complex number. The output has physical dimensions of angle.

acosech(z)
The acosech(z) function returns the hyperbolic arccosecant of z, where z may be any general dimensionless complex number. The output is dimensionless.

acosh(z)
The acosh(z) function returns the hyperbolic arccosine of z, where z may be any general dimensionless complex number. The output is dimensionless.

acot(z)
The acot(z) function returns the arccotangent of z, where z may be any general dimensionless complex number. The output has physical dimensions of angle.

acoth(z)
The acoth(z) function returns the hyperbolic arccotangent of z, where z may be any general dimensionless complex number. The output is dimensionless.

acsc(z)
The acsc(z) function returns the arccosecant of z, where z may be any general dimensionless complex number. The output has physical dimensions of angle.

acsch(z)
The acsch(z) function returns the hyperbolic arccosecant of z, where z may be any general dimensionless complex number. The output is dimensionless.

airy_ai(z)
The airy_ai(z) function returns the Airy function Ai evaluated at z, where z may be any dimensionless complex number.

airy_ai_diff(z)
The airy_ai_diff(z) function returns the first derivative of the Airy function Ai evaluated at z, where z may be any dimensionless complex number.

airy_bi(z)
The airy_bi(z) function returns the Airy function Bi evaluated at z, where z may be any dimensionless complex number.

airy_bi_diff(z)
The airy_bi_diff(z) function returns the first derivative of the Airy function Bi evaluated at z, where z may be any dimensionless complex number.

arg(z)
The arg(z) function returns the argument of the complex number z, which may have any physical dimensions. The output has physical dimensions of angle.

asec(z)
The asec(z) function returns the arcsecant of z, where z may be any general dimensionless complex number. The output has physical dimensions of angle.

asech(z)
The asech(z) function returns the hyperbolic arcsecant of z, where z may be any general dimensionless complex number. The output is dimensionless.

asin(z)
The asin(z) function returns the arcsine of z, where z may be any general dimensionless complex number. The output has physical dimensions of angle.

asinh(z)
The asinh(z) function returns the hyperbolic arcsine of z, where z may be any general dimensionless complex number. The output is dimensionless.

atan(z)
The atan(z) function returns the arctangent of z, where z may be any general dimensionless complex number. The output has physical dimensions of angle.

atan2(x, y)
The atan2(x, y) function returns the arctangent of x/y. Unlike atan(y/x), atan2(x, y) takes account of the signs of both x and y to remove the degeneracy between $(1, 1)$ and $(-1, -1)$. x and y must be real numbers, and must have matching physical dimensions.

atanh(z)
The atanh(z) function returns the hyperbolic arctangent of z, where z may be any general dimensionless complex number. The output is dimensionless.

besseli(l, x)
The besseli(l, x) function evaluates the lth regular modified spherical Bessel function at x. l must be a positive dimensionless real integer. x must be a real dimensionless number.

besselI(l, x)
The besselI(l, x) function evaluates the lth regular modified cylindrical Bessel function at x. l must be a positive dimensionless real integer. x must be a real dimensionless number.

besselj(l, x)
The besselj(l, x) function evaluates the lth regular spherical Bessel function at x. l must be a positive dimensionless real integer. x must be a real dimensionless number.

besselJ(l, x)
The besselJ(l, x) function evaluates the lth regular cylindrical Bessel function at x. l must be a positive dimensionless real integer. x must be a real dimensionless number.

besselk(l, x)
The besselk(l, x) function evaluates the lth irregular modified spherical Bessel function at x. l must be a positive dimensionless real integer. x must be a real dimensionless number.

besselK(l, x)
The besselK(l, x) function evaluates the lth irregular modified cylindrical Bessel function at x. l must be a positive dimensionless real integer. x must be a real dimensionless number.

bessely(l, x)
The bessely(l, x) function evaluates the lth irregular spherical Bessel function at x. l must be a positive dimensionless real integer. x must be a real dimensionless number.

besselY(l, x)

The besselY(l, x) function evaluates the lth irregular cylindrical Bessel function at x. l must be a positive dimensionless real integer. x must be a real dimensionless number.

beta(a, b)

The beta(a, b) function evaluates the beta function $B(a, b)$, where a and b must be dimensionless real numbers.

call(f, a)

The call(f, a) function calls the function f with the arguments contained in the list a.

ceil(x)

The ceil(x) function returns the smallest integer value greater than or equal to x, where x must be a dimensionless real number.

chr(x)

The chr(x) function returns the character with numerical ASCII code x.

classOf(x)

The classOf(x) function returns the class prototype of the object x, where x may be of any object type.

cmp(a, b)

The cmp(a, b) function returns 1 if $a > b$, -1 if $a < b$ and zero if $a = b$.

cmyk(c, m, y, k)

The cmyk(c, m, y, k) function returns a color object with the specified CMYK components in the range 0–1.

conjugate(z)

The conjugate(z) function returns the complex conjugate of the complex number z, which may have any physical dimensions.

copy(o)

The copy(o) function returns a copy of the data structure o, which may be of any object type. Nested data structures are not copied; see deepcopy(o) for this.

cos(z)

The cos(z) function returns the cosine of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

cosec(z)

The cosec(z) function returns the cosecant of z, where z may be any complex

number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

cosech(z)
The cosech(z) function returns the hyperbolic cosecant of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

cosh(z)
The cosh(z) function returns the hyperbolic cosine of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

cot(z)
The cot(z) function returns the cotangent of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

coth(z)
The coth(z) function returns the hyperbolic cotangent of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

cross(a, b)
The cross(a, b) function returns the vector cross product of the three-component vectors a and b.

csc(z)
The csc(z) function returns the cosecant of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

csch(z)
The csch(z) function returns the hyperbolic cosecant of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

deepcopy(o)
The deepcopy(o) function returns a deep copy of the data structure o, copying also any nested data structures. o may be of any object type.

degrees(x)
The degrees(x) function takes a real input which should either have physical units of angle, or be dimensionless, in which case it is assumed to be measured

in radians. The output is the dimensionless number of degrees in x.

diff_dx($e, x, step$)
The diff_dx($e, x, step$) function numerically differentiates an expression e with respect to a at x, using a step size of *step*. '`x`' can be replaced by any variable name of fewer than 16 characters, and so, for example, the `diff_dfoobar()` function differentiates an expression with respect to the variable `foobar`. The expression e may optionally be enclosed in quotes. Both x, and the output differential, may be complex numbers with any physical unit. The step size may optionally be omitted, in which case a value of 10^{-6} is used. The following example would differentiate the expression x^2 with respect to x:
`print diff_dx("x**2", 1, 1e-6)`.

ellipticintE(k)
The ellipticintE(k) function evaluates the following complete elliptic integral:

$$E(k) = \int_0^1 \sqrt{\frac{1-k^2t^2}{1-t^2}}\,\mathrm{d}t.$$

ellipticintK(k)
The ellipticintK(k) function evaluates the following complete elliptic integral:

$$K(k) = \int_0^1 \frac{\mathrm{d}t}{\sqrt{(1-t^2)(1-k^2t^2)}}.$$

ellipticintP(k, n)
The ellipticintP(k, n) function evaluates the following complete elliptic integral:

$$P(k,n) = \int_0^{\pi/2} \frac{\mathrm{d}\theta}{(1+n\sin^2\theta)(1-k^2\sin^2\theta)}.$$

erf(x)
The erf(x) function evaluates the error function at x, where x must be a dimensionless real number.

erfc(x)
The erfc(x) function evaluates the complementary error function at x, where x must be a dimensionless real number.

eval(s)
The eval(s) function evaluates the string expression s and returns the result.

exp(z)
The exp(z) function returns e^z, where z can be a complex number but must either be dimensionless or be an angle.

expint(n, x)
The expint(n, x) function evaluates the following integral:

$$\int_{t=1}^{t=\infty} \exp(-xt)/t^n \, \mathrm{d}t.$$

n must be a positive real dimensionless integer and x must be a real dimensionless number.

expm1(x)
The expm1(x) function accurately evaluates $\exp(x) - 1$, where x must be a dimensionless real number.

factors(x)
The factors(x) function returns a list of the factors of the integer x.

finite(x)
The finite(x) function returns one if x is a finite number, and zero otherwise.

floor(x)
The floor(x) function returns the largest integer value smaller than or equal to x, where x must be a dimensionless real number.

gamma(x)
The gamma(x) function evaluates the gamma function $\Gamma(x)$, where x must be a dimensionless real number.

gcd(...)
The gcd(...) function returns the greatest common divisor (a.k.a. highest common factor) of its arguments, which should be dimensionless non-zero positive integers.

globals()
The globals() function returns a dictionary of all currently-defined global variables.

gray(x)
The gray(x) function returns color object representing a shade of gray with brightness x in the range 0–1.

grey(x)
The grey(x) function returns color object representing a shade of gray with brightness x in the range 0–1.

hcf(...)
The hcf(...) function returns the highest common factor (a.k.a. greatest common divisor) of its arguments, which should be dimensionless non-zero positive integers.

heaviside(x)
The heaviside(x) function returns the Heaviside function, defined to be one for $x \geq 0$ and zero otherwise. x must be a dimensionless real number.

hsb(h, s, b)
The hsb(h, s, b) function returns color object with specified hue, saturation and brightness in the range 0–1.

hyperg_0F1(c, x)
The hyperg_0F1(c, x) function evaluates the hypergeometric function ${}_0F_1(c, x)$. All inputs must be dimensionless real numbers. For reference, the implementation used is GSL's `gsl_sf_hyperg_0F1` function.

hyperg_1F1(a, b, x)
The hyperg_1F1(a, b, x) function evaluates the hypergeometric function ${}_1F_1(a, b, x)$. All inputs must be dimensionless real numbers. For reference, the implementation used is GSL's `gsl_sf_hyperg_1F1` function.

hyperg_2F0(a, b, x)
The hyperg_2F0(a, b, x) function evaluates the hypergeometric function ${}_2F_0(a, b, x)$. All inputs must be dimensionless real numbers.For reference, the implementation used is GSL's `gsl_sf_hyperg_2F0` function.

hyperg_2F1(a, b, c, x)
The hyperg_2F1(a, b, c, x) function evaluates the hypergeometric function ${}_2F_1(a, b, c, x)$. All inputs must be dimensionless real numbers. For reference, the implementation used is GSL's `gsl_sf_hyperg_2F1` function. This implementation cannot evaluate the region $|x| < 1$.

hyperg_U(a, b, x)
The hyperg_U(a, b, x) function evaluates the hypergeometric function $U(a, b, x)$. All inputs must be dimensionless real numbers. For reference, the implementation used is GSL's `gsl_sf_hyperg_U` function.

hypot(...)
The hypot(...) function returns the quadrature sum of its arguments, $\sqrt{x^2 + y^2 + \ldots}$. Its arguments must be numerical, but may have any physical dimensions so long as they match. They can be complex numbers.

Im(z)
The Im(z) function returns the imaginary part of the complex number z, which may have any physical units. The number returned shares the same physical units as z.

int_dx(e, min, max)
The int_dx(e, min, max) function numerically integrates an expression e with respect to x between min and max. '`x`' can be replaced by any variable name of fewer than 16 characters, and so, for example, the `int_dfoobar()` function

integrates an expression with respect to the variable `foobar`. The expression e may optionally be enclosed in quotes. *min* and *max* may have any physical units, so long as they match, but must be real numbers. The output integral may be a complex number, and may have any physical dimensions. The following example would integrate the expression x^2 with respect to x between 1 m and 2 m:
`print int_dx("x**2", 1*unit(m), 2*unit(m)).`

jacobi_cn(u, m)

The jacobi_cn(u, m) function evaluates a Jacobi elliptic function; it returns the value $\cos\phi$ where ϕ is defined by the integral

$$\int_0^\phi \frac{\mathrm{d}\theta}{\sqrt{1 - m\sin^2\theta}}.$$

jacobi_dn(u, m)

The jacobi_dn(u, m) function evaluates a Jacobi elliptic function; it returns the value $\sqrt{1 - m\sin^2\theta}$ where ϕ is defined by the integral

$$\int_0^\phi \frac{\mathrm{d}\theta}{\sqrt{1 - m\sin^2\theta}}.$$

jacobi_sn(u, m)

The jacobi_sn(u, m) function evaluates a Jacobi elliptic function; it returns the value $\sin\phi$ where ϕ is defined by the integral

$$\int_0^\phi \frac{\mathrm{d}\theta}{\sqrt{1 - m\sin^2\theta}}.$$

lambert_W0(x)

The lambert_W0(x) function evaluates the principal real branch of the Lambert W function, for which $W > -1$ when $x < 0$.

lambert_W1(x)

The lambert_W1(x) function evaluates the secondary real branch of the Lambert W function, for which $W < -1$ when $x < 0$.

lcm(...)

The lcm(...) function returns the lowest common multiple of its arguments, which should be dimensionless positive integers.

ldexp(x, y)

The ldexp(x, y) function returns x times 2^y for integer y, where both x and y must be real.

legendreP(l, x)
The legendreP(l, x) function evaluates the lth Legendre polynomial at x, where l must be a positive dimensionless real integer and x must be a real dimensionless number.

legendreQ(l, x)
The legendreQ(l, x) function evaluates the lth Legendre function at x, where l must be a positive dimensionless real integer and x must be a real dimensionless number.

len(o)
The len(o) function returns the length of the object o. The may be the length of a string, or the number of entries in a compound data type.

ln(z)
The ln(z) function returns the natural logarithm of z, where z may be any complex dimensionless number.

locals()
The locals() function returns a dictionary of all currently-defined local variables in the present scope.

log(z)
The log(z) function returns the natural logarithm of z, where z may be any complex dimensionless number.

log10(z)
The log10(z) function returns the logarithm to base 10 of z, where z may be any complex dimensionless number.

logn(x, n)
The logn(x, n) function returns the logarithm of x to base n.

lrange([f],l,[s])
The lrange([f],l,[s]) function returns a vector of numbers between f and l with uniform multiplicative spacing s. If not specified, $f = 1$ and $s = 2$. If two arguments are specified, these are interpreted as f and l. The arguments f and l may have any physical units, so long as they match. s must be a dimensionless number.

matrix(...)
The matrix(...) function creates a new matrix object. See `types.matrix`.

max(...)
The max(...) function returns the highest-valued of its arguments, which may be of any object type and may have any physical dimensions, so long as they match. If either input is complex, the input with the larger magnitude is returned. If a single vector or list object is supplied, the highest-valued item in the vector or

list is returned.

min(...)
The min(...) function returns the lowest-valued of its arguments, where may be of any object type and may have any physical dimensions, so long as they match. If either input is complex, the input with the smaller magnitude is returned. If a single vector or list object is supplied, the lowest-valued item in the vector or list is returned.

mod(x, y)
The mod(x, y) function returns the remainder of x/y, where x and y may have any physical dimensions so long as they match but must both be real.

module(...)
The module(...) function creates a new module object. See `types.module`.

open(x[,y])
The open(x[,y]) function opens the file x with string access mode y, and returns a file handle object.

ord(s)
The ord(s) function returns the ASCII code of the first character of the string s.

ordinal(x)
The ordinal(x) function returns an ordinal string, for example, "1st", "2nd" or "3rd", for any positive dimensionless real number x.

pow(x, y)
The pow(x, y) function returns x to the power of y, where x and y may both be complex numbers and x may have any physical dimensions but y must be dimensionless. It not not permitted for y to be complex if x is not dimensionless, since this would lead to an output with complex physical dimensions.

prime(x)
The prime(x) function returns one if floor(x) is a prime number and zero otherwise.

primeFactors(x)
The primeFactors(x) function returns a list of the prime factors of the integer x.

radians(x)
The radians(x) function takes a real input which should either have physical units of angle, or be dimensionless, in which case it is assumed to be measured in degrees. The output is the dimensionless number of radians in x.

raise(e, s)
The raise(e, s) function raises the exception e, with error string s. e should be an exception object; s should be an error message string.

range([f],l,[s])
The range([f],l,[s]) function returns a vector of uniformly-spaced numbers between f and l, with stepsize s. If not specified, $f = 0$ and $s = 1$. If two arguments are specified, these are interpreted as f and l. The arguments may have any physical units, so long as they match.

Re(z)
The Re(z) function returns the real part of the complex number z, which may have any physical units. The number returned shares the same physical units as z.

rgb(r, g, b)
The rgb(r, g, b) function returns a color object with specified RGB components in the range 0–1.

romanNumeral(x)
The romanNumeral(x) function returns the Roman numeral representing the number x, for any positive dimensionless real input less than 10,000.

root(z, n)
The root(z, n) function returns the nth root of z. z may be any complex number, and may have any physical dimensions. n must be a dimensionless integer. When complex arithmetic is enabled, and whenever z is positive, this function is entirely equivalent to `pow(z,1/n)`. However, when z is negative and complex arithmetic is disabled, the expression `pow(z,1/n)` may not be evaluated, since it will in general have a small imaginary part for any finite-precision floating-point representation of $1/n$. The expression `root(z,n)`, on the other hand, may be evaluated under such conditions, providing that n is an odd integer.

round(x)
The round(x) function rounds the value x to the nearest integer. If x is exactly halfway between integers, it is rounded away from zero. x must be a dimensionless real number.

sec(z)
The sec(z) function returns the secant of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

sech(z)
The sech(z) function returns the hyperbolic secant of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

sgn(x)
The sgn(x) function returns 1 if x is greater than zero, -1 if x is less than zero, and 0 if x equals zero.

sin(z)
The sin(z) function returns the sine of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

sinc(z)
The sinc(z) function returns the sinc function $\sin(z)/z$ for any complex number z, which may either be dimensionless, in which case it is understood to be measured in radians, or have physical dimensions of angle. The result is dimensionless.

sinh(z)
The sinh(z) function returns the hyperbolic sine of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

sqrt(z)
The sqrt(z) function returns the square root of z, which may be any complex number and may have any physical dimensions.

sum(...)
The sum(...) function returns the sum of its arguments, which be of any object type, and may have any physical units, so long as it is possible to add them together.

tan(z)
The tan(z) function returns the tangent of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

tanh(z)
The tanh(z) function returns the hyperbolic tangent of z, where z may be any complex number and must either have physical dimensions of angle or be a dimensionless number, in which case it is understood to be measured in radians.

texify(s)
The texify(s) function takes a string representation of an algebraic expression as its input, e.g. "`(x/3)**2`", and returns a latex representation of it.

texifyText(s)
The texifyText(s) function returns a string of latex text corresponding to the supplied text string, with any reserved characters escaped.

tophat(x, σ)
The tophat(x, σ) function returns one if $|x| \leq |\sigma|$, and zero otherwise. Both inputs must be real, but may have any physical dimensions so long as they match.

typeOf(o)
The typeOf(o) function returns the type of the object o.

unit(...)
The unit(...) function multiplies a number by a physical unit. The string inside the brackets should consist of a string of the names of physical units, multiplied together with the `*` operator, divided using the `/` operator, or raised by numeric powers using the `^` operator. The list may be commenced with a numeric constant, for example: `unit(2*m^2/s)`.

vector(...)
The vector(...) function creates a new vector object. See `types.vector`.

zernike(n, m, r, ϕ)
The zernike(n, m, r, ϕ) function evaluates the Zernike polynomial $Z_n^m(r, \phi)$, where m and n are non-negative integers with $n \geq m$, r is the radial coordinate in the range $0 < r < 1$ and ϕ is the azimuthal coordinate.

zernikeR(n, m, r)
The zernikeR(n, m, r) function evaluates the radial Zernike polynomial $R_n^m(r)$, where m and n are non-negative integers with $n \geq m$ and r is the radial coordinate in the range $0 < r < 1$.

zeta(x)
The zeta(x) function evaluates the Riemann zeta function for any dimensionless number x.

12.0.1 The `ast` module

The `ast` module contains specialist functions for astronomy and cosmology.

ast.Lcdm_age(H_0,Ω_M,Ω_Λ)
The ast.Lcdm_age(H_0,Ω_M,Ω_Λ) function returns the current age of the Universe in a standard Λ_CDM cosmology with specified values for Hubble's constant, Ω_M and Ω_Λ. Hubble's constant should be specified either with physical units of recession velocity per unit distance, or as a dimensionless number, assumed to have implicit units of km/s/Mpc. Suitable input values for a standard cosmology are: $H_0 = 70$, $\Omega_\mathrm{M} = 0.27$ and $\Omega_\Lambda = 0.73$. For more details, see David W. Hogg's short article *Distance measures in cosmology*, available online at: http://arxiv.org/abs/astro-ph/9905116.

ast.Lcdm_angscale(z,H_0,Ω_M,Ω_Λ)
The ast.Lcdm_angscale(z,H_0,Ω_M,Ω_Λ) function returns the angular scale of the

sky at a redshift of z in a standard $\Lambda_{\rm CDM}$ cosmology. For details, see the `ast.Lcdm_age()` function above. The returned value has dimensions of distance per unit angle.

ast.Lcdm_DA(z,H_0,$\Omega_{\rm M}$,Ω_Λ)

The ast.Lcdm_DA(z,H_0,$\Omega_{\rm M}$,Ω_Λ) function returns the angular size distance of objects at a redshift of z in a standard $\Lambda_{\rm CDM}$ cosmology. For details, see the `ast.Lcdm_age()` function above. The returned value has dimensions of distance.

ast.Lcdm_DL(z,H_0,$\Omega_{\rm M}$,Ω_Λ)

The ast.Lcdm_DL(z,H_0,$\Omega_{\rm M}$,Ω_Λ) function returns the luminosity distance of objects at a redshift of z in a standard $\Lambda_{\rm CDM}$ cosmology. For details, see the `ast.Lcdm_age()` function above. The returned value has dimensions of distance.

ast.Lcdm_DM(z,H_0,$\Omega_{\rm M}$,Ω_Λ)

The ast.Lcdm_DM(z,H_0,$\Omega_{\rm M}$,Ω_Λ) function returns the proper motion distance of objects at a redshift of z in a standard $\Lambda_{\rm CDM}$ cosmology. For details, see the `ast.Lcdm_age()` function above. The returned value has dimensions of distance.

ast.Lcdm_t(z,H_0,$\Omega_{\rm M}$,Ω_Λ)

The ast.Lcdm_t(z,H_0,$\Omega_{\rm M}$,Ω_Λ) function returns the lookback time to objects at a redshift of z in a standard $\Lambda_{\rm CDM}$ cosmology. For details, see the `ast.Lcdm_age()` function above. The returned value has dimensions of time. To find the age of the Universe at a redshift of z, this value should be subtracted from the output of the `ast.Lcdm_age()` function.

ast.Lcdm_z(t,H_0,$\Omega_{\rm M}$,Ω_Λ)

The ast.Lcdm_z(t,H_0,$\Omega_{\rm M}$,Ω_Λ) function returns the redshift corresponding to a lookback time of t in a standard $\Lambda_{\rm CDM}$ cosmology. For details, see the `ast.Lcdm_age()` function above. The returned value is dimensionless.

ast.moonphase(d)

The ast.moonphase(d) function returns the phase of the Moon, with dimensions of angle, at the time of the supplied date object d. If d is numerical, it is assumed to be a Unix time.

ast.sidereal_time(d)

The ast.sidereal_time(d) function returns the sidereal time at Greenwich, with dimensions of angle, at the time of the supplied date object d. If d is numerical, it is assumed to be a Unix time. The returned sidereal time is equal to the right ascension of the stars which are transiting the Greenwich meridian at that time. This function uses the expression for sidereal time adopted in 1982 by the International Astronomical Union (IAU), and which is reproduced in Chapter 12 of Jean Meeus' book *Astronomical Algorithms* (1998).

12.0.2 The `colors` module

The `colors` module contains color objects representing all of Pyxplot's default colors. It also contains the following functions:

colors.spectrum(*spec*, *norm*)
The colors.spectrum(*spec*, *norm*) function returns a color representation of the spectrum *spec*, normalised to brightness *norm*. *spec* should be a function object that takes a single input (wavelength) with units of length, and may return an output with arbitrary units.

colors.wavelength(λ,*norm*)
The colors.wavelength(λ,*norm*) function returns a color representation of monochromatic light at wavelength λ, normalised to brightness *norm*. A value of *norm* = 1 is recommended for plotting the complete span of the electromagnetic spectrum without colors clipping to white.

12.0.3 The `exceptions` module

The exceptions module contains the following objects of type `exception`:
`assertion`, `file`, `generic`, `interrupt`, `key`, `namespace`, `numerical`, `overflow`, `range`, `syntax`, `type`, `unit`.
To raise an exception with one of these types, the `raise` function should be called:

raise(e, s)
The raise(e, s) function raises the exception e, with error string s. e should be an exception object; s should be an error message string.

12.0.4 The `fractals` module

fractals.julia(z,z_c,m)
The fractals.julia(z,z_c,m) function tests whether the point z in the complex plane lies within the Julia set associated with the point z_c in the complex plane. The expression $z_{n+1} = z_n^2 + z_c$ is iterated until either $|z_n| > 2$, in which case the iteration is deemed to have diverged, or until m iterations have been exceeded, in which case it is deemed to have remained bounded. The number of iterations required for divergence is returned, or m is returned if the iteration remained bounded – i.e. the point lies within the numerical approximation to the Julia set.

fractals.mandelbrot(z,m)
The fractals.mandelbrot(z,m) function tests whether the point z in the complex plane lies within the Mandelbrot set. The expression $z_{n+1} = z_n^2 + z_0$ is iterated until either $|z_n| > 2$, in which case the iteration is deemed to have diverged, or until m iterations have been exceeded, in which case it is deemed to have remained bounded. The number of iterations required for divergence is returned,

or m is returned if the iteration remained bounded – i.e. the point lies within the numerical approximation to the Mandelbrot set.

12.0.5 The `os` module

os.chdir(x)
The os.chdir(x) function changes working directory to x, which should be a string.

os.getcwd()
The os.getcwd() function returns the path of the current working directory.

os.getegid()
The os.getegid() function returns the effective group id of the Pyxplot process.

os.geteuid()
The os.geteuid() function returns the effective user id of the Pyxplot process.

os.getgid()
The os.getgid() function returns the group id of the Pyxplot process.

os.gethomedir()
The os.gethomedir() function returns the path of the user's home directory.

os.gethostname()
The os.gethostname() function returns the system's host name.

os.getlogin()
The os.getlogin() function returns the system login of the user.

os.getpgrp()
The os.getpgrp() function returns the process group id of the Pyxplot process.

os.getpid()
The os.getpid() function returns the process id of the Pyxplot process.

os.getppid()
The os.getppid() function returns the parent process id of the Pyxplot process.

os.getrealname()
The os.getrealname() function returns the user's real name.

os.getuid()
The os.getuid() function returns the user id of the Pyxplot process.

os.glob(x)
The os.glob(x) function returns a list of files which match the supplied wildcard x, which should be a string.

os.popen(x,[y])
The os.popen(x,[y]) function opens a pipe to the command x with string access mode y, and returns a file handle object.

os.stat(x)
The os.stat(x) function returns a dictionary of information about the file x, which should be a string filename.

os.stderr
The os.stderr function is a file handle for the Pyxplot process's `stderr` stream.

os.stdin
The os.stdin function is a file handle for the Pyxplot process's `stdin` stream.

os.stdout
The os.stdout function is a file handle for the Pyxplot process's `stdout` stream.

os.system(x)
The os.system(x) function executes a command in a subshell.

os.tmpfile()
The os.tmpfile() function returns a file handle for a temporary file.

os.uname()
The os.uname() function returns a dictionary of information about the operating system.

12.0.6 The `os.path` module

os.path.atime(x)
The os.path.atime(x) function returns a date object representing the time of the last access of the file with pathname x.

os.path.ctime(x)
The os.path.ctime(x) function returns a date object representing the time of the last status change to the file with pathname x.

os.path.exists(x)
The os.path.exists(x) function returns a boolean flag indicating whether a file with pathname x exists.

os.path.expanduser(x)
The os.path.expanduser(x) function returns its argument with ∼s indicating home directories expanded.

os.path.filesize(x)
The os.path.filesize(x) function returns the size, with physical dimensions of bytes, of the file with pathname x.

os.path.join(...)
The os.path.join(...) function joins a series of strings intelligently into a pathname.

os.path.mtime(x)
The os.path.mtime(x) function returns a date object representing the time of the last modification of the file with pathname x.

12.0.7 The `phy` module

The `phy` module contains a selection of physical constants, listed in Chapter 14. It also contains the following functions:

phy.Bv(ν, T)
The phy.Bv(ν, T) function returns the power emitted by a blackbody of temperature T at frequency ν per unit area, per unit solid angle, per unit frequency. T should have physical dimensions of temperature, or be a dimensionless number, in which case it is understood to be a temperature in Kelvin. ν should have physical dimensions of frequency, or be a dimensionless number, in which case it is understood to be a frequency measured in Hertz. The output has physical dimensions of power per unit area per unit solid angle per unit frequency.

phy.Bvmax(T)
The phy.Bvmax(T) function returns the frequency at which the function Bv(ν, T) reaches its maximum, as calculated by the Wien Displacement Law. The inputs are subject to the same constraints as above.

12.0.8 The `random` module

The `random` module contains function for generating random samples from probability distributions:

random.binomial(p, n)
The random.binomial(p, n) function returns a random sample from a binomial distribution with n independent trials and a success probability p. n must be a real positive dimensionless integer. p must be a dimensionless number in the range $0 \leq p \leq 1$.

random.chisq(ν)
The random.chisq(ν) function returns a random sample from a χ-squared distri-

bution with ν degrees of freedom, where ν must be a real positive dimensionless integer.

random.gaussian(σ)
The random.gaussian(σ) function returns a random sample from a Gaussian (normal) distribution of standard deviation σ and centred upon zero. σ must be real, but may have any physical units. The returned random sample shares the physical units of σ.

random.lognormal(ζ, σ)
The random.lognormal(ζ, σ) function returns a random sample from the log normal distribution centred on ζ, and of width σ. σ must be a real positive dimensionless number. ζ must be real, but may have any physical units. The returned random sample shares the physical units of ζ.

random.poisson(n)
The random.poisson(n) function returns a random integer from a Poisson distribution with mean n, where n must be a real positive dimensionless number.

random.random()
The random.random() function returns a random real number between 0 and 1.

random.tdist(ν)
The random.tdist(ν) function returns a random sample from a t-distribution with ν degrees of freedom, where ν must be a real positive dimensionless integer.

12.0.9 The `stats` module

The `stats` module contains statistical functions:

stats.binomialCDF(k, p, n)
The stats.binomialCDF(k, p, n) function evaluates the probability of getting fewer than or exactly k successes out of n trials in a binomial distribution with success probability p. k and n must be positive real integers. p must be a real number in the range $0 \leq p \leq 1$.

stats.binomialPDF(k, p, n)
The stats.binomialPDF(k, p, n) function evaluates the probability of getting k successes out of n trials in a binomial distribution with success probability p. k and n must be positive real integers. p must be a real number in the range $0 \leq p \leq 1$.

stats.chisqCDF(x, ν)
The stats.chisqCDF(x, ν) function returns the cumulative probability density at x in a χ-squared distribution with ν degrees of freedom. ν must be a positive

real dimensionless integer. x must be a positive real dimensionless number.

stats.chisqCDFi(P, ν)
The stats.chisqCDFi(P, ν) function returns the point x at which the cumulative probability density in a χ-squared distribution with ν degrees of freedom is P. ν must be a positive real dimensionless integer. P must be a real number in the range $0 \leq p \leq 1$.

stats.chisqPDF(x, ν)
The stats.chisqPDF(x, ν) function returns the probability density at x in a χ-squared distribution with ν degrees of freedom. ν must be a positive real dimensionless integer. x must be a positive real dimensionless number.

stats.gaussianCDF(x, σ)
The stats.gaussianCDF(x, σ) function evaluates the Gaussian cumulative distribution function of standard deviation σ at x. The distribution is centred upon $x = 0$. x and σ must both be real, but may have any physical dimensions so long as they match.

stats.gaussianCDFi(x, σ)
The stats.gaussianCDFi(x, σ) function evaluates the inverse Gaussian cumulative distribution function of standard deviation σ at x. The distribution is centred upon $x = 0$. x and σ must both be real, but may have any physical dimensions so long as they match.

stats.gaussianPDF(x, σ)
The stats.gaussianPDF(x, σ) function evaluates the Gaussian probability density function of standard deviation σ at x. The distribution is centred upon $x = 0$. x and σ must both be real, but may have any physical dimensions so long as they match.

stats.lognormalCDF(x, ζ, σ)
The stats.lognormalCDF(x, ζ, σ) function evaluates the log normal cumulative distribution function of standard deviation σ, centred upon ζ, at x. σ must be real, positive and dimensionless. x and ζ must both be real, but may have any physical dimensions so long as they match.

stats.lognormalCDFi(x, ζ, σ)
The stats.lognormalCDFi(x, ζ, σ) function evaluates the inverse log normal cumulative distribution function of standard deviation σ, centred upon ζ, at x. σ must be real, positive and dimensionless. x and ζ must both be real, but may have any physical dimensions so long as they match.

stats.lognormalPDF(x, ζ, σ)
The stats.lognormalPDF(x, ζ, σ) function evaluates the log normal probability density function of standard deviation σ, centred upon ζ, at x. σ must be real, positive and dimensionless. x and ζ must both be real, but may have any physical dimensions so long as they match.

stats.poissonCDF(x, μ)
The stats.poissonCDF(x, μ) function returns the probability of getting $\leq x$ from a Poisson distribution with mean μ, where μ must be real, positive and dimensionless and x must be real and dimensionless.

stats.poissonPDF(x, μ)
The stats.poissonPDF(x, μ) function returns the probability of getting x from a Poisson distribution with mean μ, where μ must be real, positive and dimensionless and x must be a real dimensionless integer.

stats.tdistCDF(x, ν)
The stats.tdistCDF(x, ν) function returns the cumulative probability density at x in a t-distribution with ν degrees of freedom. ν must be a positive real dimensionless integer. x must be a positive real dimensionless number.

stats.tdistCDFi(P, ν)
The stats.tdistCDFi(P, ν) function returns the point x at which the cumulative probability density in a t-distribution with ν degrees of freedom is P. ν must be a positive real dimensionless integer. P must be a real number in the range $0 \leq p \leq 1$.

stats.tdistPDF(x, ν)
The stats.tdistPDF(x, ν) function returns the probability density at x in a t-distribution with ν degrees of freedom. ν must be a positive real dimensionless integer. x must be a positive real dimensionless number.

12.0.10 The `time` module

The `time` module contains functions for handling objects of type `date`. For more information about manipulating times and dates in Pyxplot, see Section 4.11. Many of the functions below take an optional timezone string as their final argument. This should be specified in the form `Europe/London`, `America/New_York` or `Australia/Perth`, as used by the `tz database`. A complete list of available timezones can be found here: http://en.wikipedia.org/wiki/List_of_tz_database_time_zones. If universal time is used, the timezone may be specified as `UTC`.

time.fromCalendar($year, month, day, hour, min, sec, < timezone >$)
The time.fromCalendar($year, month, day, hour, min, sec, < timezone >$) function creates a date object from the specified calendar date. See also the `set calendar` and `set timezone` commands to change the current calendar and timezone.

time.fromJD(t)
The time.fromJD(t) function creates a date object from the specified numerical Julian date.

time.fromMJD(t)
The time.fromMJD(t) function creates a date object from the specified numerical modified Julian date.

time.fromUnix(t)
The time.fromUnix(t) function creates a date object from the specified numerical Unix time.

time.interval(t_2, t_1)
The time.interval(t_2, t_1) function returns the numerical time interval between date objects t_1 and t_2, with physical units of time.

time.intervalStr($t_2, t_1, format$)
The time.intervalStr($t_2, t_1, format$) function returns a string representation of the time interval elapsed between the first and second supplied date objects. The third input is used to control the format of the output, with the following tokens being substituted for:

Token	Value
`%%`	A literal % sign.
`%d`	The number of days elapsed, modulo 365.
`%D`	The number of days elapsed.
`%h`	The number of hours elapsed, modulo 24.
`%H`	The number of hours elapsed.
`%m`	The number of minutes elapsed, modulo 60.
`%M`	The number of minutes elapsed.
`%s`	The number of seconds elapsed, modulo 60.
`%S`	The number of seconds elapsed.
`%Y`	The number of years elapsed.

time.now()
The time.now() function creates a date object representing the current time.

time.sleep(t)
The time.sleep(t) function sleeps for t seconds, or for time period t if it has dimensions of time.

time.sleepUntil(t)
The time.sleepUntil(t) function sleeps until the specified date and time. Its argument should be a date object.

time.string($t, < format >, < timezone >$)

The time.string($t, < format >, < timezone >$) function returns a string representation of the specified date object t. The second input is optional, and may be used to control the format of the output. If no format string is provided, then the format

```
"%a %Y %b %d %H:%M:%S"
```

is used. In such format strings, the following tokens are substituted for various parts of the date:

Token	Value
`%%`	A literal % sign.
`%a`	Three-letter abbreviated weekday name.
`%A`	Full weekday name.
`%b`	Three-letter abbreviated month name.
`%B`	Full month name.
`%C`	Century number, e.g. 21 for the years 2000-2100.
`%d`	Day of month.
`%H`	Hour of day, in range 00-23.
`%I`	Hour of day, in range 01-12.
`%k`	Hour of day, in range 0-23.
`%l`	Hour of day, in range 1-12.
`%m`	Month number, in range 01-12.
`%M`	Minute, in range 00-59.
`%p`	Either `am` or `pm`.
`%S`	Second, in range 00-59.
`%y`	Last two digits of year number.
`%Y`	Year number.

12.0.11 The `types` module

The `types` module contains prototype objects for each of Pyxplot's data types. Each may be called like a function to create a new object of the specified type:

types.boolean(...)

The types.boolean(...) prototype takes any of the following combinations of arguments:

- **none** – returns false.
- **any object** – returns false for zero, an empty string, a null object, or an empty data structure. Otherwise returns true.

types.color(...)
The types.color(...) prototype takes any of the following combinations of arguments:

- **none** – returns black.
- **a color** – returns a copy of that color.
- **a number** – returns the color at the specified position in the present palette.

types.date(...)
The types.date(...) prototype takes any of the following combinations of arguments:

- **none** – returns the current time.
- **a date** – returns a copy of that date.

types.dictionary(...)
The types.dictionary(...) prototype takes any of the following combinations of arguments:

- **none** – returns an empty dictionary.
- **a dictionary** – returns a deep copy of the supplied dictionary.

types.exception(...)
The types.exception(...) prototype takes any of the following combinations of arguments:

- **a string** – returns an exception type with the specified name.
- **an exception** – returns a copy of the supplied exception type.

types.fileHandle(...)
The types.fileHandle(...) prototype takes any of the following combinations of arguments:

- **none** – returns a file handle of a temporary file.
- **a file handle** – returns a copy of the supplied file handle.

types.function(...)
The types.function(...) prototype takes any of the following combinations of arguments:

- Cannot be called. Functions should be created using the syntax `f(x)=...`

types.instance(...)

The types.instance(...) prototype takes any of the following combinations of arguments:

- **none** – returns an empty module.
- **a module or instance** – returns a deep copy of the supplied module or instance.

types.list(...)

The types.list(...) prototype takes any of the following combinations of arguments:

- **none** – returns an empty list.
- **a list** – returns a deep copy of the supplied list.
- **a vector** – returns a list representation of the supplied vector.
- **a list of arguments** returns its arguments as a list.

types.matrix(...)

The types.matrix(...) prototype takes any of the following combinations of arguments:

- **none** – returns a one-by-one zero matrix.
- **a pair of numbers** – returns a zero matrix of the specified dimensions. The first number is the number of rows, and the second the number of columns.
- **a matrix** – returns a copy of the supplied matrix.
- **one or more vectors, or a list of vectors** – converts the supplied vector(s) into the **columns** of a new matrix object.
- **one or more lists, or a list of lists** – converts the supplied list(s) into the **rows** of a new matrix object, providing all the elements are numerical and have matching units.

types.module(...)

The types.module(...) prototype takes any of the following combinations of arguments:

- **none** – returns an empty module.
- **a module** – returns a deep copy of the supplied module.

types.null(...)

The types.null(...) prototype takes any of the following combinations of arguments:

- **none** – returns a null object.

types.number(...)

The types.number(...) prototype takes any of the following combinations of arguments:

- **none** – returns zero.
- **a number** – returns that number.
- **a boolean** – returns zero or one.
- **a string** – converts the string to a number if it is in a valid format, or returns an error if not.

types.string(...)

The types.string(...) prototype takes any of the following combinations of arguments:

- **none** – returns an empty string.
- **any object** – returns a string representation of the object.

types.type(...)

The types.type(...) prototype takes any of the following combinations of arguments:

- Cannot be called. New data types cannot be created except by instantiating modules.

types.vector(...)

The types.vector(...) prototype takes any of the following combinations of arguments:

- **none** – returns the vector [0].
- **a number** – returns a zero vector with the specified number of dimensions.
- **a vector** – returns a copy of the supplied vector.
- **a list** – converts the supplied list to a vector, providing all of its elements are numerical and have consistent units.
- **a list of arguments** returns its arguments as a vector, providing they are all numerical and have consistent units.

Chapter 13

List of data types

The following is a list of Pyxplot's data types:

- boolean
- color
- date
- dictionary
- exception
- fileHandle
- function
- instance
- list
- matrix
- module
- null
- number
- string
- type
- vector

Each of these data types has a prototype object in the module `types`, which can be called like a function to create a new object of the type. See Section 12.0.11 for details of the arguments accepted by each prototype.

All objects in Pyxplot have methods that can be called on them, using the generic syntax:

```
object.methodName(arguments)
```

Some methods are common to all objects. For example, all objects have a method `str()` which produces a string representation of the object, as used by the `print` command. They also all have a `methods()` method, which returns a list of the names of all of the methods available for the object. For example:

```
pyxplot> pi.type()
<data type: number>
pyxplot> "hello world".methods
methods() returns a list of the methods of an object.
pyxplot> [3,2,1].len()
3
pyxplot> time.fromCalendar(2000,1,1,0,0,0).toUnix()
946684800
```

As the above examples show, printing a method object returns brief documentation about it. The sections below list the methods of each data type.

13.1 Methods common to all data types

class()
The class() method returns the class prototype of an object.

contents()
The contents() method returns a list of all the methods and internal variables of an object.

data()
The data() method returns a list of all the internal variables (not methods) of an object.

methods()
The methods() method returns a list of the methods of an object.

str()
The str() method returns a string representation of an object.

type()
The type() method returns the type of an object.

13.2 The `boolean` type

The `boolean` type has no methods other than those common to all types. Objects of type `boolean` have no methods other than those common to all types.

13.3 The color type

componentsCMYK()
The componentsCMYK() method returns a vector CMYK representation of a color.

componentsHSB()
The componentsHSB() method returns a vector HSB representation of a color.

componentsRGB()
The componentsRGB() method returns a vector RGB representation of a color.

toCMYK()
The toCMYK() method returns color object containing a CMYK representation of a color.

toHSB()
The toHSB() method returns color object containing an HSB representation of a color.

toRGB()
The toRGB() method returns color object containing an RGB representation of a color.

13.4 The date type

For more information about manipulating dates in Pyxplot, see Section 4.11. For more information about manipulating times and dates in Pyxplot, see Section 4.11. Many of the methods listed below take an optional timezone string as their final argument. This should be specified in the form `Europe/London`, `America/New_York` or `Australia/Perth`, as used by the `tz database`. A complete list of available timezones can be found here: http://en.wikipedia.org/wiki/List_of_tz_database_time_zones. If universal time is used, the timezone may be specified as `UTC`.

str(< *format* >, < *timezone* >)
The str(< *format* >, < *timezone* >) method converts a date object to a string with an optional format string supplied as an argument (see the `time.string()` function).

toDayOfMonth(< *timezone* >)
The toDayOfMonth(< *timezone* >) method returns the day of the month of a date object in the current calendar.

toDayWeekName(< *timezone* >**)**
The toDayWeekName(< *timezone* >) method returns the name of the day of the week of a date object.

toDayWeekNum(< *timezone* >**)**
The toDayWeekNum(< *timezone* >) method returns the day of the week (1–7) of a date object.

toHour(< *timezone* >**)**
The toHour(< *timezone* >) method returns the integer hour component (0–23) of a date object.

toJD()
The toJD() method converts a date object to a numerical Julian date.

toMinute(< *timezone* >**)**
The toMinute(< *timezone* >) method returns the integer minute component (0–59) of a date object.

toMJD()
The toMJD() method converts a date object to a modified Julian date.

toMonthName(< *timezone* >**)**
The toMonthName(< *timezone* >) method returns the name of the month in which a date object falls.

toMonthNum(< *timezone* >**)**
The toMonthNum(< *timezone* >) method returns the number (1–12) of the month in which a date object falls.

toSecond(< *timezone* >**)**
The toSecond(< *timezone* >) method returns the seconds component (0–60) of a date object, including the non-integer component.

toUnix()
The toUnix() method converts a date object to a Unix time.

toYear(< *timezone* >**)**
The toYear(< *timezone* >) method returns the year in which a date object falls in the current calendar.

13.5 The `dictionary` type

delete(s**)**
The delete(s) method deletes any element with string key s from the dictionary.

hasKey(x)
The hasKey(x) method returns a boolean indicating whether the key x exists in the dictionary.

items()
The items() method returns a list of the [key,value] pairs in a dictionary.

keys()
The keys() method returns a list of the keys defined in a dictionary.

len()
The len() method returns the number of entries in a dictionary.

values()
The values() method returns a list of the values in a dictionary.

13.6 The `exception` type

raise(x)
The raise(x) method raises an exception with error message string x.

13.7 The `fileHandle` type

close()
The close() method closes a file handle.

dump(x)
The dump(x) method stores a typeable ASCII representation of the object x to a file. Note that this method has no checking for recursive hierarchical data structures.

eof()
The eof() method returns a boolean flag to indicate whether the end of a file has been reached.

flush()
The flush() method flushes any buffered data which has not yet physically been written to a file.

getPos()
The getPos() method returns a file handle's current position in a file.

isOpen()
The isOpen() method returns a boolean flag indicating whether a file is open.

read()
The read() method returns the contents of a file as a string.

readline()
The readline() method returns a single line of a file as a string.

readlines()
The readlines() method returns the lines of a file as a list of strings.

setPos(x)
The setPos(x) method sets a file handle's current position in a file.

write(x)
The write(x) method writes the string x to a file.

13.8 The `function` type

Objects of type `function` have no methods other than those common to all types.

13.9 The `instance` type

delete(s)
The delete(s) method deletes any element with string key s from the instance.

hasKey(x)
The hasKey(x) method returns a boolean indicating whether the key x exists in the instance.

items()
The items() method returns a list of the [key,value] pairs in a instance.

keys()
The keys() method returns a list of the keys defined in a instance.

len()
The len() method returns the number of entries in a instance.

values()
The values() method returns a list of the values in a instance.

13.10 The `list` type

append(x)
The append(x) method appends the object x to a list and returns the new list.

count(x)
The count(x) method returns the number of items in a list that equal x.

extend(x)
The extend(x) method appends the members of a list or vector x to the operand and returns the new list.

filter(f)
The filter(f) method takes a pointer to a function of one argument, $f(a)$. It calls the function for every element of the list, and returns a new list of those elements for which $f(a)$ tests true.

index(x)
The index(x) method returns the index of the first element of a list that equals x, or -1 if no elements match.

insertn, x)
The insertn, x) method inserts the number x into a list at position n, and returns the new list.

len()
The len() method returns the number of elements in a list.

map(f)
The map(f) method takes a pointer to a function of one argument, $f(a)$. It calls the function for every element of the list, and returns a list of the results.

max()
The max() method returns the highest-valued item in a list.

min()
The min() method returns the lowest-valued item in a list.

pop()
The pop() method returns the last item in a list, and removes it from the list.

reduce(f)
The reduce(f) method takes a pointer to a function of two arguments. It first calls $f(a, b)$ on the first two elements of the list, and then continues through the list calling $f(a, b)$ on the result and the next item in the list. The final result is

returned.

reverse()
The reverse() method reverses the order of the members of a list, and returns the new list.

sort()
The sort() method sorts the members of a list into ascending order, and returns the new list.

sortOn(f)
The sortOn(f) method sorts the members of a list using the user-supplied function $f(a, b)$ to determine the sort order. f should return 1, 0 or -1 depending whether $a > b$, $a = b$ or $a < b$.

sortOnElement(n)
The sortOnElement(n) method sorts a list of lists on the nth element of each sublist. If n is negative, it counts from the final item of each list, $n = -1$ being the last item.

vector()
The vector() method returns the elements in a list as a vector.

13.11 The `matrix` type

det()
The det() method returns the determinant of a square matrix.

diagonal()
The diagonal() method returns a boolean indicating whether a matrix is diagonal.

eigenvalues()
The eigenvalues() method returns a vector containing the eigenvalues of a square symmetric matrix.

eigenvectors()
The eigenvectors() method returns a list of the eigenvectors of a square symmetric matrix.

inv()
The inv() method returns the inverse of a square matrix.

size()
The size() method returns the dimensions of a matrix.

symmetric()
The symmetric() method returns a boolean indicating whether a matrix is symmetric.

transpose()
The transpose() method returns the transpose of a matrix.

13.12 The `module` type

delete(s)
The delete(s) method deletes any element with string key s from the module.

hasKey(x)
The hasKey(x) method returns a boolean indicating whether the key x exists in the module.

items()
The items() method returns a list of the [key,value] pairs in a module.

keys()
The keys() method returns a list of the keys defined in a module.

len()
The len() method returns the number of entries in a module.

values()
The values() method returns a list of the values in a module.

13.13 The `null` type

Objects of type `null` have no methods other than those common to all types.

13.14 The `number` type

Objects of type `number` have no methods other than those common to all types.

13.15 The `string` type

append(x)
The append(x) method appends the string x to the end of a string.

beginsWith(x)
The beginsWith(x) method returns a boolean indicating whether a string begins with the substring x.

endsWith(x)
The endsWith(x) method returns a boolean indicating whether a string ends with the substring x.

find(x)
The find(x) method returns the position of the first occurrence of x in a string, or -1 if it is not found.

findAll(x)
The findAll(x) method returns a list of the positions where the substring x occurs in a string.

isalnum()
The isalnum() method returns a boolean indicating whether all of the characters of a string are alphanumeric.

isalpha()
The isalpha() method returns a boolean indicating whether all of the characters of a string are alphabetic.

isdigit()
The isdigit() method returns a boolean indicating whether all of the characters of a string are numeric.

len()
The len() method returns the length of a string.

lower()
The lower() method converts a string to lowercase.

lstrip()
The lstrip() method strips whitespace off the beginning of a string.

split()
The split() method returns a list of all the whitespace-separated words in a string.

splitOn(...)
The splitOn(...) method splits a string whenever it encounters any of the substrings supplied as arguments, and returns a list of the split string segments.

strip()
The strip() method strips whitespace off the beginning and end of a string.

rstrip()
The rstrip() method strips whitespace off the end of a string.

upper()
The upper() method converts a string to uppercase.

13.16 The `type` type

Objects of type `type` have no methods other than those common to all types.

13.17 The `vector` type

append(x)
The append(x) method appends the number x to a vector and returns the new vector.

extend(x)
The extend(x) method appends the members of a list or vector x to the operand and returns the new vector.

filter(f)
The filter(f) method takes a pointer to a function of one argument, $f(a)$. It calls the function for every element of the vector, and returns a new vector of those elements for which $f(a)$ tests true.

insert(n, x)
The insert(n, x) method inserts the number x into a vector at position n, and returns the new vector.

len()
The len() method returns the number of elements in a vector.

list()
The list() method returns the elements in a vector as a list.

map(f)
The map(f) method takes a pointer to a function of one argument, $f(a)$. It calls the function for every element of the vector, and returns a vector of the results.

norm()
The norm() method returns the quadrature sum of the elements in a vector.

reduce(f)
The reduce(f) method takes a pointer to a function of two arguments. It first calls $f(a, b)$ on the first two elements of the vector, and then continues through the vector calling $f(a, b)$ on the result and the next item in the vector. The final result is returned.

reverse()
The reverse() method reverses the order of the elements of a vector, and returns the new vector.

sort()
The sort() method sorts the elements of a vector into ascending order, and returns the new vector.

Chapter 14

List of physical constants

The following table lists all of the physical constants which are defined by default in Pyxplot:

Name	Description	Approximate Value
`e`	e	2.7182818
`euler`	The Euler constant	0.57721566
`false`	Boolean truth value	0
`goldenRatio`	The golden ratio	1.618034
`i`	The square-root of -1	i
`phy.alpha`	The fine-structure constant	0.0072973525
`phy.c`	The speed of light	$299792458\,\mathrm{m/s}$
`phy.epsilon_0`	The permittivity of free space	$8.85418782 \times 10^{-12}\,\mathrm{F/m}$
`phy.G`	The gravitational constant	$6.673 \times 10^{-11}\,\mathrm{m^3/kg/s^2}$
`phy.g`	The mean terrestrial acceleration due to gravity	$9.80665\,\mathrm{m/s^2}$
`phy.h`	The Planck constant	$6.62606896 \times 10^{-34}\,\mathrm{J\,s}$
`phy.hbar`	The Planck constant / 2π	$1.05457163 \times 10^{-34}\,\mathrm{J\,s}$
`phy.kB`	The Boltzmann constant	$1.3806504 \times 10^{-23}\,\mathrm{J/K}$
`phy.Lsun`	The luminosity of the Sun	$3.839 \times 10^{26}\,\mathrm{W}$
`phy.Msun`	The mass of the Sun	$1.98892 \times 10^{30}\,\mathrm{kg}$
`phy.mu_0`	The permeability of free space	$1.25663706 \times 10^{-6}\,\mathrm{N/A^2}$
`phy.mu_b`	The Bohr magneton	$9.27400899 \times 10^{-24}\,\mathrm{J/T}$
`phy.m_e`	The mass of the electron	$9.10938188 \times 10^{-31}\,\mathrm{kg}$
`phy.m_muon`	The mass of the muon	$1.88353109 \times 10^{-28}\,\mathrm{kg}$
`phy.m_n`	The mass of the neutron	$1.67492716 \times 10^{-27}\,\mathrm{kg}$
`phy.m_p`	The mass of the proton	$1.67262158 \times 10^{-27}\,\mathrm{kg}$
`phy.m_u`	The unified mass constant	$1.66053878 \times 10^{-27}\,\mathrm{kg}$
`phy.NA`	Avogadro's number	$6.02214199 \times 10^{23}\,\mathrm{mol^{-1}}$
`phy.q`	The fundamental charge	$1.60217649 \times 10^{-19}\,\mathrm{C}$
`phy.R`	The gas constant	$8.314472\,\mathrm{J/K/mol}$
`phy.Rsun`	The radius of the Sun	$695500000\,\mathrm{m}$
`phy.Ry`	The Rydberg constant	$10973732\,\mathrm{m^{-1}}$

Name	Description	Approximate Value
`phy.sigma`	The Stefan-Boltzmann constant	$5.67040047 \times 10^{-8}\,\mathrm{kg/s^3/K^4}$
`pi`	π	3.1415927
`true`	Boolean truth value	1
`version`	Pyxplot version string	"0.9.2"

Chapter 15

List of physical units

The following table lists all of the physical units which Pyxplot recognises by default. Each unit may be referred to by either a long or an abbreviated name, and both of these have singular and plural forms. Some units also have further alternative names: for example, units such as the `meter` which are spelt differently in British English may be spelt in either way.

Name Full sing.	pl.	Abbrev sing.	pl.	Unit of
acre	acres	acre	acres	area
ampere	amperes	A	A	current
				Also known as the amp and the amps.
angstrom	angstroms	ang	ang	length
arcminute	arcminutes	arcmin	arcmins	angle
arcsecond	arcseconds	arcsec	arcsecs	angle
are	ares	are	ares	area
astronomical_unit	astronomical_units	AU	AU	length
atmosphere	atmospheres	atm	atms	pressure
bar	bars	bar	bars	pressure
barleycorn	barleycorns	barleycorn	barleycorns	length
barn	barns	barn	barns	area
barye	baryes	Ba	Ba	pressure
bath	baths	bath	baths	volume
becquerel	becquerel	Bq	Bq	frequency
billion_electronvolts	billion_electronvolts	BeV	BeV	energy
bit	bits	bit	bits	information_content
British Thermal Unit	British Thermal Units	BTU	BTU	energy
bushel_UK	bushels_UK	bushel_UK	bushels_UK	volume (UK imperial)
bushel_US	bushels_US	bushel_US	bushels_US	volume (US customary)
byte	bytes	B	B	information_content
cable	cables	cable	cables	length
calorie	calories	cal	cal	energy
candela	candelas	cd	cd	light_intensity
candlepower	candlepower	candlepower	candlepower	light_intensity

Name				
Full		**Abbrev**		**Unit of**
sing.	**pl.**	**sing.**	**pl.**	
carat	carats	CD	CDs	mass
centimetre	centimetres	cm	cm	length
chain	chains	chain	chains	length
clo	clos	clo	clos	thermal_insulation
coulomb	coulombs	C	C	charge
cubic_centimetre	cubic_centimetres	cubic_cm	cubic_cm	volume
cubic_foot	cubic_feet	cubic_ft	cubic_ft	volume
cubic_inch	cubic_inches	cubic_in	cubic_in	volume
cubic_metre	cubic_metres	cubic_m	cubic_m	volume
cubit	cubits	cubit	cubits	length
cup_US	cups_US	cup_US	cups_US	volume (US customary)
day	days	day	days	time
decimetre	decimetres	dm	dm	length
degree	degrees	deg	deg	angle
degree_celsius	degrees_celsius	oC	oC	temperature
	Also known as the degree_centigrade, the degrees_centigrade, the centigrade and the celsius.			
degree_fahrenheit	degrees_fahrenheit	oF	oF	temperature
				Also known as the fahrenheit.
dioptre	dioptres	dioptre	dioptres	lens_power
drachm	drachms	drachm	drachms	mass
dyne	dynes	dyn	dyn	force
earth_mass	earth_masses	Mearth	Mearth	mass
earth_radius	earth_radii	Rearth	Rearth	length
electronvolt	electronvolts	eV	eV	energy
erg	ergs	erg	erg	energy

Name Full sing.	pl.	Abbrev sing.	pl.	Unit of
euro	euros	euro	euros	cost
farad	farad	F	F	capacitance
fathom	fathoms	fathom	fathoms	length
firkin_UK_ale	firkins_UK_ale	firkin_UK_ale	firkins_UK_ale	volume
firkin_wine	firkins_wine	firkin_UK_wine	firkins_UK_wine	volume
fluid_ounce_UK	fluid_ounce_UK	fl_oz_UK	fl_oz_UK	volume (UK imperial)
fluid_ounce_US	fluid_ounce_US	fl_oz_US	fl_oz_US	volume (US customary)
foot	feet	ft	ft	length
furlong	furlongs	furlong	furlongs	length
gallon_UK	gallons_UK	gallon_UK	gallons_UK	volume (UK imperial)
gallon_US	gallons_US	gallon_US	gallons_US	volume (US customary)
gauss	gauss	G	G	magnetic_field
gibibit	gibibits	Gib	Gib	information_content
gibibyte	gibibytes	GiB	GiB	information_content
grain	grains	grain	grains	mass
gram	grams	g	g	mass
gramme	grammes	g	g	mass
gray	gray	Gy	Gy	radiation_dose
hectare	hectares	hectare	hectares	area
henry	henry	H	H	inductance
hertz	hertz	Hz	Hz	frequency
homer	homers	homer	homers	volume
horsepower	horsepower	horsepower	horsepower	power
hour	hours	hr	hr	time
hundredweight_UK	hundredweight_UK	cwt_UK	cwt_UK	mass (UK imperial)

Name Full sing.	pl.	Abbrev sing.	pl.	Unit of
hundredweight_US	hundredweight_US	cwt_US	cwt_US	mass (US customary)
inch	inches	in	in	length
inch_of_mercury	inches_of_mercury	inHg	inHg	pressure
inch_of_water	inches_of_water	inAq	inAq	pressure
jansky	janskys	Jy	Jy	flux_density
joule	joules	J	J	energy
jupiter_mass	jupiter_masses	Mjupiter	Mjupiter	mass
			Also known as the Mjove and the Mjovian.	
jupiter_radius	jupiter_radii	Rjupiter	Rjupiter	length
			Also known as the Rjove and the Rjovian.	
katal	katals	kat	kat	catalytic_activity
kayser	kaysers	kayser	kaysers	wavenumber
kelvin	kelvin	K	K	temperature
kibibit	kibibits	Kib	Kib	information_content
kibibyte	kibibytes	KiB	KiB	information_content
kilderkin_UK_ale	kilderkins_UK_ale	kilderkin_UK_ale	kilderkins_UK_ale	volume
kilogram	kilograms	kg	kg	mass
kilowatt_hour	kilowatt_hours	kWh	kWh	energy
knot	knots	kn	kn	velocity
light_year	light_years	lyr	lyr	length
link	links	link	links	length
litre	litres	l	l	volume
long_ton	long_tons	ton	tons	mass (UK imperial)
lumen	lumens	lm	lm	power
lunar_distance	lunar_distances	lunar_distance	lunar_distances	length

Name Full sing.	pl.	Abbrev sing.	pl.	Unit of
lux	luxs	lx	lx	power
maxwell	maxwell	Mx	Mx	magnetic_flux
mebibit	mebibits	Mib	Mib	information_content
mebibyte	mebibytes	MiB	MiB	information_content
metre	metres	m	m	length
mho	mhos	mho	mhos	conductance
mile	miles	mi	mi	length
mile_per_hour	miles_per_hour	mph	mph	velocity
mina	minas	mina	minas	mass
minute	minutes	min	min	time
mole	moles	mol	mol	moles
nautical_mile	nautical_miles	nautical_mile	nautical_miles	length
newton	newtons	N	N	force
ohm	ohms	ohm	ohms	resistance
ounce	ounces	oz	oz	mass
parsec	parsecs	pc	pc	length
parts_per_billion	parts_per_billion	ppb	ppb	dimensionlessness
parts_per_million	parts_per_million	ppm	ppm	dimensionlessness
pascal	pascals	Pa	Pa	pressure
percent	percent	percent	percent	dimensionlessness
perch	perches	perch	perches	length
pica	picas	pica	picas	length
pint_UK	pints_UK	pint_UK	pints_UK	volume (UK imperial)
pint_US	pints_US	pint_US	pints_US	volume (US customary)
planck_charge	planck_charges	Q_planck	Q_planck	charge

Name Full sing.	pl.	Abbrev sing.	pl.	Unit of
planck_current	planck_current	I_planck	I_planck	current
planck_energy	planck_energy	E_planck	E_planck	energy
planck_force	planck_force	F_planck	F_planck	force
planck_impedence	planck_impedence	Z_planck	Z_planck	resistance
planck_length	planck_lengths	L_planck	L_planck	length
planck_mass	planck_masses	M_planck	M_planck	mass
planck_momentum	planck_momentum	p_planck	p_planck	momentum
planck_power	planck_power	P_planck	P_planck	power
planck_temperature	planck_temperature	Theta_planck	Theta_planck	temperature
planck_time	planck_times	T_planck	T_planck	time
planck_voltage	planck_voltage	V_planck	V_planck	potential
point	points	pt	pt	length
poise	poises	P	P	viscosity
pole	poles	pole	poles	length
pound	pounds	lb	lbs	mass
pound_force	pounds_force	lbf	lbf	force
pound_per_square_inch	pounds_per_square_inch	psi	psi	pressure
quart_UK	quarts_UK	quart_UK	quarts_UK	volume (UK imperial)
quart_US	quarts_US	quart_US	quarts_US	volume (US customary)
radian	radians	rad	rad	angle
rankin	rankin	R	R	temperature
revolution	revolutions	rev	rev	angle
rod	rods	rod	rods	length
roman_league	roman_leagues	roman_league	roman_leagues	length
roman_mile	roman_miles	roman_mile	roman_miles	length

Name Full sing.	pl.	Abbrev sing.	pl.	Unit of
rood	roods	rood	roods	area
second	seconds	s	s	time
				Also known as the sec and the secs.
shekel	shekels	shekel	shekels	mass
short_ton	short_tons	short_ton	short_tons	mass (US customary)
siemens	siemens	S	S	conductance
sievert	sieverts	Sv	Sv	radiation_dose
slug	slugs	slug	slugs	mass
sol	sols	sol	sols	time
solar_luminosity	solar_luminosities	Lsun	Lsun	power
				Also known as the Lsolar.
solar_mass	solar_masses	Msun	Msun	mass
				Also known as the Msolar.
solar_radius	solar_radii	Rsun	Rsun	length
				Also known as the Rsolar.
square_centimetre	square_centimetres	sq_cm	sq_cm	area
square_degree	square_degrees	sqdeg	sqdeg	solidangle
square_foot	square_feet	sq_ft	sq_ft	area
square_inch	square_inches	sq_in	sq_in	area
square_kilometre	square_kilometres	sq_km	sq_km	area
square_metre	square_metres	sq_m	sq_m	area
square_mile	square_miles	sq_mi	sq_mi	area
steradian	steradians	sterad	sterad	solidangle
stone	stone	stone	stone	mass
tablespoon	tablespoons	tablespoon	tablespoons	volume

Name		Abbrev		Unit of
Full sing.	pl.	sing.	pl.	
talent	talents	talent	talents	mass
teaspoon	teaspoons	teaspoon	teaspoons	volume
tesla	tesla	T	T	magnetic_field
therm	therms	therm	therms	energy
tog	togs	tog	togs	thermal_insulation
tonne	tonnes	t	t	mass
		Also known as the metric_tonne and the metric_tonnes.		
troy_ounce	troy_ounces	oz_troy	oz_troy	mass
volt	volts	V	V	potential
watt	watts	W	W	power
weber	weber	Wb	Wb	magnetic_flux
week	weeks	week	weeks	time
yard	yards	yd	yd	length
year	years	yr	yr	time

The following units of **angle** are recognised:
the `arcminute`, the `arcsecond`, the `degree`, the `radian` and the `revolution`.

The following units of **area** are recognised:
the `acre`, the `are`, the `barn`, the `hectare`, the `rood`, the `square_centimetre`, the `square_foot`, the `square_inch`, the `square_kilometre`, the `square_metre` and the `square_mile`.

The following units of **capacitance** are recognised:
the `farad`.

The following units of **catalytic activity** are recognised:
the `katal`.

The following units of **charge** are recognised:
the `coulomb` and the `planck_charge`.

The following units of **conductance** are recognised:
the `mho` and the `siemens`.

The following units of **cost** are recognised:
the `euro`.

The following units of **current** are recognised:
the `ampere` and the `planck_current`.

The following units of **dimensionlessness** are recognised:
the `parts_per_billion`, the `parts_per_million` and the `percent`.

The following units of **energy** are recognised:
the `British Thermal Unit`, the `billion_electronvolts`, the `calorie`, the `electronvolt`, the `erg`, the `joule`, the `kilowatt_hour`, the `planck_energy` and the `therm`.

The following units of **flux density** are recognised:
the `jansky`.

The following units of **force** are recognised:
the `dyne`, the `newton`, the `planck_force` and the `pound_force`.

The following units of **frequency** are recognised:
the `becquerel` and the `hertz`.

The following units of **inductance** are recognised:
the `henry`.

The following units of **information content** are recognised:

the `bit`, the `byte`, the `gibibit`, the `gibibyte`, the `kibibit`, the `kibibyte`, the `mebibit` and the `mebibyte`.

The following units of **length** are recognised:
the `angstrom`, the `astronomical_unit`, the `barleycorn`, the `cable`, the `centimetre`, the `chain`, the `cubit`, the `decimetre`, the `earth_radius`, the `fathom`, the `foot`, the `furlong`, the `inch`, the `jupiter_radius`, the `light_year`, the `link`, the `lunar_distance`, the `metre`, the `mile`, the `nautical_mile`, the `parsec`, the `perch`, the `pica`, the `planck_length`, the `point`, the `pole`, the `rod`, the `roman_league`, the `roman_mile`, the `solar_radius` and the `yard`.

The following units of **lens power** are recognised:
the `dioptre`.

The following units of **light intensity** are recognised:
the `candela`[1] and the `candlepower`.

The following units of **magnetic field** are recognised:
the `gauss` and the `tesla`.

The following units of **magnetic flux** are recognised:
the `maxwell` and the `weber`.

The following units of **mass** are recognised:
the `carat`, the `drachm`, the `earth_mass`, the `grain`, the `gram`, the `gramme`, the `hundredweight_UK`, the `hundredweight_US`, the `jupiter_mass`, the `kilogram`, the `long_ton`, the `mina`, the `ounce`, the `planck_mass`, the `pound`, the `shekel`, the `short_ton`, the `slug`, the `solar_mass`, the `stone`, the `talent`, the `tonne` and the `troy_ounce`.

The following units of **moles** are recognised:
the `mole`.

The following units of **momentum** are recognised:
the `planck_momentum`.

The following units of **potential** are recognised:
the `planck_voltage` and the `volt`.

The following units of **power** are recognised:
the `horsepower`, the `lumen`, the `lux`, the `planck_power`, the `solar_luminosity` and the `watt`.

The following units of **pressure** are recognised:

[1] In the SI system, the candela is a base unit. However, its definition is 1/683 W/sterad, and since Pyxplot recognises the radian as a base unit, the candela is implemented as a derived unit.

the `atmosphere`, the `bar`, the `barye`, the `inch_of_mercury`, the `inch_of_water`, the `pascal` and the `pound_per_square_inch`.

The following units of **radiation dose** are recognised:
the `gray` and the `sievert`.

The following units of **resistance** are recognised:
the `ohm` and the `planck_impedence`.

The following units of **solidangle** are recognised:
the `square_degree` and the `steradian`.

The following units of **temperature** are recognised:
the `degree_celsius`, the `degree_fahrenheit`, the `kelvin`, the `planck_temperature` and the `rankin`.

The following units of **thermal insulation** are recognised:
the `clo` and the `tog`.

The following units of **time** are recognised:
the `day`, the `hour`, the `minute`, the `planck_time`, the `second`, the `sol`, the `week` and the `year`.

The following units of **velocity** are recognised:
the `knot` and the `mile_per_hour`.

The following units of **viscosity** are recognised:
the `poise`.

The following units of **volume** are recognised:
the `bath`, the `bushel_UK`, the `bushel_US`, the `cubic_centimetre`, the `cubic_foot`, the `cubic_inch`, the `cubic_metre`, the `cup_US`, the `firkin_UK_ale`, the `firkin_wine`, the `fluid_ounce_UK`, the `fluid_ounce_US`, the `gallon_UK`, the `gallon_US`, the `homer`, the `kilderkin_UK_ale`, the `litre`, the `pint_UK`, the `pint_US`, the `quart_UK`, the `quart_US`, the `tablespoon` and the `teaspoon`.

The following units of **wavenumber** are recognised:
the `kayser`.

Chapter 16

List of paper sizes

The following table lists all of the named paper sizes which Pyxplot recognises:

Name	h/**mm**	w/**mm**
2a0	1681	1189
4a0	2378	1681
a0	1189	840
a1	840	594
a10	37	26
a2	594	420
a3	420	297
a4	297	210
a5	210	148
a6	148	105
a7	105	74
a8	74	52
a9	52	37
b0	1414	999
b1	999	707
b10	44	31
b2	707	499
b3	499	353
b4	353	249
b5	249	176
b6	176	124
b7	124	88
b8	88	62
b9	62	44
c0	1296	917
c1	917	648
c10	40	28
c2	648	458
c3	458	324
c4	324	229
c5	229	162
c6	162	114
c7	114	81
c8	81	57
c9	57	40
crown	508	381
demy	572	445
double_demy	889	597
elephant	711	584
envelope_dl	110	220
executive	267	184
foolscap	330	203
government_letter	267	203
international_businesscard	85	53

Name	h/**mm**	w/**mm**
japanese_b0	1435	1015
japanese_b1	1015	717
japanese_b10	44	31
japanese_b2	717	507
japanese_b3	507	358
japanese_b4	358	253
japanese_b5	253	179
japanese_b6	179	126
japanese_b7	126	89
japanese_b8	89	63
japanese_b9	63	44
japanese_kiku4	306	227
japanese_kiku5	227	151
japanese_shiroku4	379	264
japanese_shiroku5	262	189
japanese_shiroku6	188	127
large_post	533	419
ledger	432	279
legal	356	216
letter	279	216
medium	584	457
monarch	267	184
post	489	394
quad_demy	1143	889
quarto	254	203
royal	635	508
statement	216	140
swedish_d0	1542	1090
swedish_d1	1090	771
swedish_d10	48	34
swedish_d2	771	545
swedish_d3	545	385
swedish_d4	385	272
swedish_d5	272	192
swedish_d6	192	136
swedish_d7	136	96
swedish_d8	96	68
swedish_d9	68	48
swedish_e0	1241	878
swedish_e1	878	620
swedish_e10	38	27
swedish_e2	620	439
swedish_e3	439	310
swedish_e4	310	219
swedish_e5	219	155
swedish_e6	155	109
swedish_e7	109	77

Name	h/**mm**	w/**mm**
swedish_e8	77	54
swedish_e9	54	38
swedish_f0	1476	1044
swedish_f1	1044	738
swedish_f10	46	32
swedish_f2	738	522
swedish_f3	522	369
swedish_f4	369	261
swedish_f5	261	184
swedish_f6	184	130
swedish_f7	130	92
swedish_f8	92	65
swedish_f9	65	46
swedish_g0	1354	957
swedish_g1	957	677
swedish_g10	42	29
swedish_g2	677	478
swedish_g3	478	338
swedish_g4	338	239
swedish_g5	239	169
swedish_g6	169	119
swedish_g7	119	84
swedish_g8	84	59
swedish_g9	59	42
swedish_h0	1610	1138
swedish_h1	1138	805
swedish_h10	50	35
swedish_h2	805	569
swedish_h3	569	402
swedish_h4	402	284
swedish_h5	284	201
swedish_h6	201	142
swedish_h7	142	100
swedish_h8	100	71
swedish_h9	71	50
tabloid	432	279
us_businesscard	89	51

Chapter 17

Color tables

Figures 17.1, 17.2 and 17.3 show the default named colors which Pyxplot recognises. In addition to using these colors in statements such as

```
plot 'data' with color red
```

it is also possible to make custom colors using the `rgb(r,g,b)`, `cmyk(c,m,y,k)`, `gray(g)` and `hsb(h,s,b)` functions, whose inputs should be in the range 0–1. For example:

```
plot 'data' with color rgb(0.8,0.8,0.2)

myColor = cmyk(0.2,0.8,0.8,0.1)
plot 'data' with color myColor
```

These figures also exclude the 100 shades of gray which Pyxplot recognises, which are named from `gray00` (black) to `gray99` (almost white). These shades of gray may also be spelt in the UK English form `grey??`.

Figure 17.1: A list of the named colors which Pyxplot recognises, sorted alphabetically. The numerous shades of gray which it recognises are not shown.

Figure 17.2: A list of the named colors which Pyxplot recognises, sorted by hue. The numerous shades of gray which it recognises are not shown.

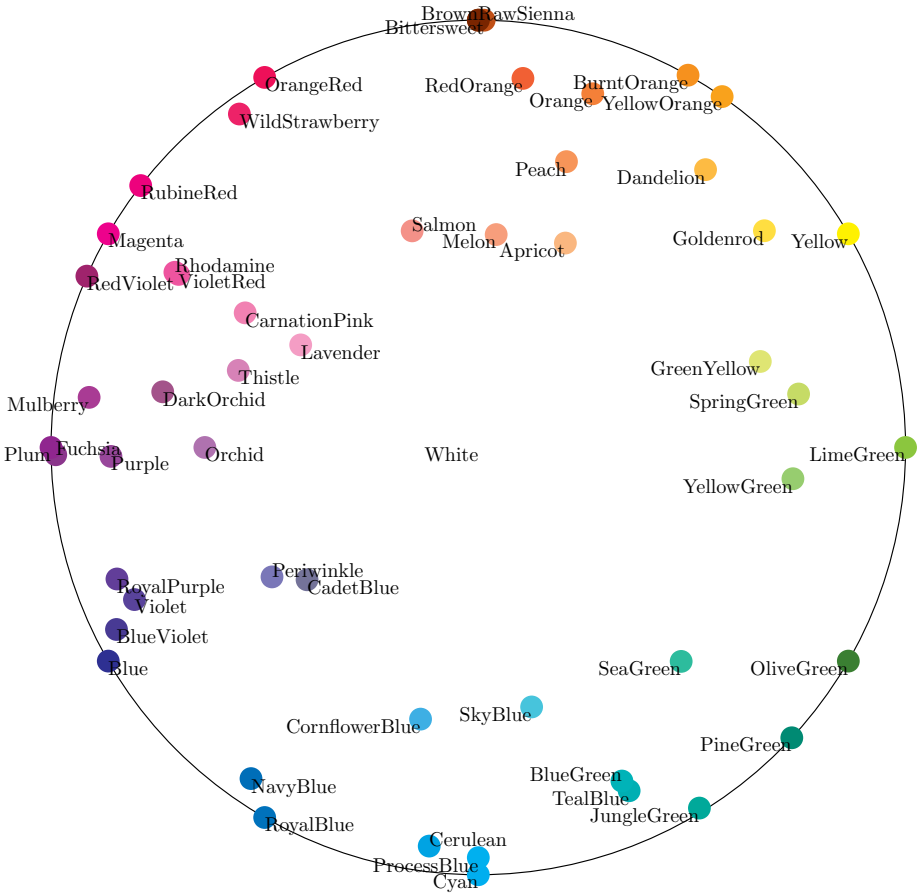


Figure 17.3: The named colors which Pyxplot recognises, arranged in HSB color space, with the brightness axis orientated into the page. Some colors are not shown as they lie too close to others.

Chapter 18

Line and point types

The tables in this chapter show the appearances of each of the numbered line, point and star types available in the `lines`, `points` and `stars` plot styles respectively.

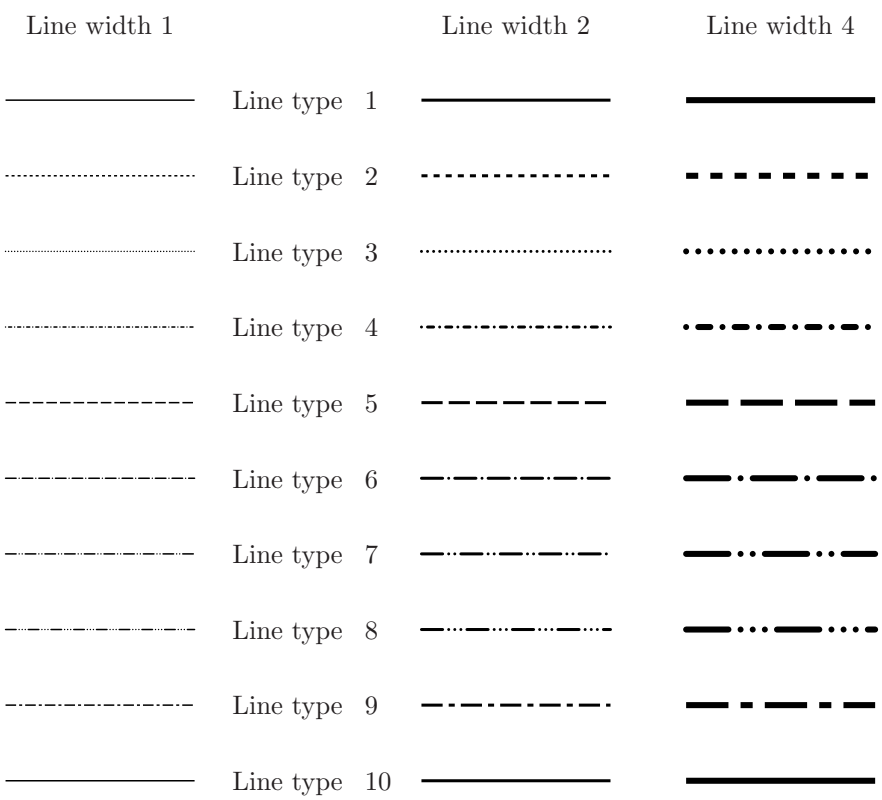


Table 18.1: The numbered line types available in the `lines` plot style.

× Point type 1	■ Point type 16	● Point type 31
□ Point type 2	● Point type 17	▲ Point type 32
○ Point type 3	▲ Point type 18	♦ Point type 33
△ Point type 4	♦ Point type 19	⬢ Point type 34
◊ Point type 5	⬢ Point type 20	⬟ Point type 35
⬡ Point type 6	⬟ Point type 21	▼ Point type 36
⬠ Point type 7	▼ Point type 22	▶ Point type 37
▽ Point type 8	▶ Point type 23	◀ Point type 38
▷ Point type 9	◀ Point type 24	★ Point type 39
◁ Point type 10	⋆ Point type 25	⧗ Point type 40
★ Point type 11	⧗ Point type 26	⋈ Point type 41
⧗ Point type 12	⋈ Point type 27	✳ Point type 42
⋈ Point type 13	⋎ Point type 28	● Point type 43
⋏ Point type 14	✳ Point type 29	● Point type 44
+ Point type 15	■ Point type 30	

Table 18.2: The numbered point types available in the `points` plot style.

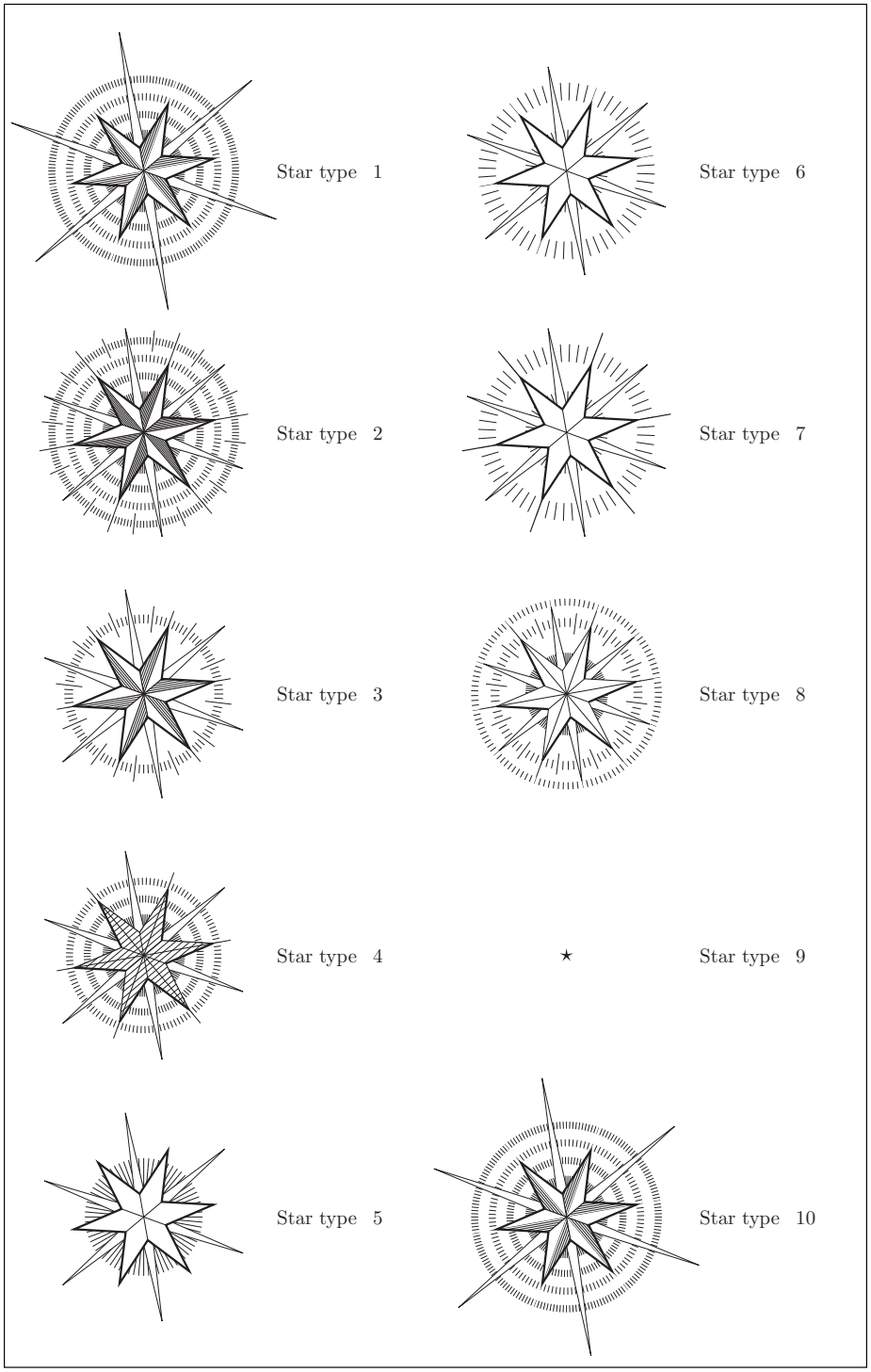


Table 18.3: The numbered star types available in the `stars` plot style.

Chapter 19

Configuring Pyxplot

In Parts I and II, we encountered numerous configuration options within Pyxplot which can controlled using the `set` command. There are times, however, when many plots are wanted in a homogeneous style, or when a single plot is repeatedly generated, when it is desirable to change the default set of configuration options with which Pyxplot starts up, in order to avoid having to repeated enter a large number of `set` commands. In this chapter, we describe the use of configuration files to program Pyxplot's default state.

19.1 Configuration files

Configuration files for Pyxplot have the filename `.pyxplotrc`, and may be placed either in a user's home directory, in which case they globally affect all of that user's Pyxplot sessions, or in particular directories, in which case they only affect Pyxplot sessions which are instantiated with that particular directory as the current working directory. When configuration files are present in both locations, both are read; settings found in the `.pyxplotrc` file in the current working directory take precedence over those found in the user's home directory. Configuration files are read only once, upon startup, and subsequent changes to the configuration files do not affect copies of Pyxplot which are already running.

Changes to settings made in configuration files affect not only the values that these settings have upon startup; they also changes the values to which the `unset` command returns settings. Thus, whilst the command

```
unset multiplot
```

ordinarily turns off multiplot mode, it may turn it on if a configuration file contains the line

```
multiPlot=on
```

When colored terminal output is enabled, the color-coding of the `show` command also reflects the current default configuration: settings which match their default values are shown in green[1] whilst those settings which have been changed from their default values are shown in amber[2].

[1] This color can be changed using the `color_rep` setting in a configuration file.
[2] This color can be changed using the `color_wrn` setting in a configuration file.

Configuration files should take the form of a series of sections, each headed by a section heading enclosed in square brackets. Each section heading should be followed by a series of settings, which often take the form of

```
setting_name = value
```

In *most* cases, neither the setting name nor the value are case sensitive.

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

- `colors` – contains a single setting `palette`, which should be set to a comma-separated list of colors which should make up the palette used to plot datasets. The first will be called color 1 in Pyxplot, the second color 2, etc. A list of recognised color names is given in Section 19.4.
- `filters` – can be used to define input filters which should be used for certain file types (see Section 5.1).
- `functions` – contains user-defined function definitions which become pre-defined in Pyxplot's mathematical environment, for example

  ```
  sinc(x) = sin(x)/(x)
  ```

- `latex` – contains a single setting `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 also changed using the `set preamble` command.
- `script` – can contain a list of `set` commands, using the same syntax which would be used to enter them at a Pyxplot command prompt. This section provides an alternative and more general way of controlling the settings which can be changed in the `settings` section. Note that this section may only contain instances of the `set` command; other Pyxplot commands may not be used. The `set` command's `item` modifier may not be used.
- `settings` – contains settings similar to those which can be set with the `set` command. A complete list is given in Section 19.3.2 below.
- `styling` – contains settings which control various detailed aspects of the graphical output which Pyxplot produces. These settings cannot be accessed by any other means.
- `terminal` – contains settings for altering the behaviour and appearance of Pyxplot's interactive terminal. These cannot be changed with the `set` command, and can only be controlled via configuration files. A complete list of the available settings is given in Section 19.3.4.
- `units` – can be used to define new physical units for use in Pyxplot's mathematical environment.
- `variables` – contains variable definitions, in the format

  ```
  variable = value
  ```

  Any variables defined in this section will be pre-defined in Pyxplot's mathematical environment upon startup.

19.2 An example configuration file

The following configuration file represents Pyxplot's default configuration, and provides a useful index to all of the settings which are available. In subsequent sections, we describe the effect of each setting in detail.

```
[settings]
aspect = auto
axesColor = black
axisUnitStyle = ratio
backup = off
bar = 1.0
binOrigin = auto
binWidth = auto
boxFrom = auto
boxWidth = auto
calendarIn = British
calendarOut = British
clip = off
colKey = on
colKeyPos = Right
color = on
contours = 12
c1Range_log = false
c1Range_max = 0
c1Range_max_Auto = true
c1Range_min = 0
c1Range_min_Auto = true
c1Range_renorm = true
c1Range_reverse = false
c2Range_log = false
c2Range_max = 0
c2Range_max_auto = true
c2Range_min = 0
c2Range_min_auto = true
c2Range_renorm = true
c2Range_reverse = false
c3Range_log = false
c3Range_max = 0
c3Range_max_auto = true
c3Range_min = 0
c3Range_min_auto = true
c3Range_renorm = true
c3Range_reverse = false
c4Range_log = false
c4Range_max = 0
c4Range_max_auto = true
c4Range_min = 0
c4Range_min_auto = true
c4Range_renorm = true
```

```
c4Range_reverse = false
dataStyle = Points
display = on
dpi = 300
fontSize = 1
funcStyle = Lines
grid = off
gridAxisX = 1
gridAxisY = 1
gridAxisZ = 1
gridMajColor = grey70
gridMinColor = grey85
key = on
keyColumns = 0
keyPos = top right
key_Xoff = 0.0
key_Yoff = 0.0
landscape = off
lineWidth = 1.0
multiPlot = off
numComplex = off
numDisplay = natural
numErr = on
numSF = 8
originX = 0.0
originY = 0.0
output =
paperHeight = 297
paperName = a4
paperWidth = 210
pointLineWidth = 1.0
pointSize = 1.0
samples = 250
samples_method = nearestNeighbor
samples_x_auto = false
samples_x = 40
samples_y_auto = false
samples_y = 40
termAntiAlias = on
termEnlarge = off
termInvert = off
termTransparent = off
termType = X11_singleWindow
textColor = black
textHAlign = left
textVAlign = bottom
title =
title_Xoff = 0.0
title_Yoff = 0.0
uRange_log = false
```

```
uRange_max = 1.0
uRange_min = 0.0
vRange_log = false
vRange_max = 1.0
vRange_min = 0.0
tRange_log = false
tRange_max = 1.0
tRange_min = 0.0
unitAbbrev = on
unitAngleDimless = on
unitPrefix = on
unitScheme = si
width = 8.0
view_xy = 60
view_yz = 30
zAspect = auto

[terminal]
color = on
color_err = red
color_rep = green
color_wrn = amber
splash = on

[styling]
arrow_headAngle = 45
arrow_headSize = 1.0
arrow_headBackIndent = 0.2
axes_lineWidth = 1.0
axes_majTickLen = 1.0
axes_minTickLen = 1.0
axes_separation = 1.0
axes_textGap = 1.0
colorScale_margin = 1.0
colorScale_width = 1.0
grid_majLineWidth = 1.0
grid_minLineWidth = 0.5
baseline_lineWidth = 1.0
baseline_pointSize = 1.0

[variables]
pi = 3.14159265358979

[colors]
palette = black, red, blue, magenta, cyan, brown, salmon, gray,
green, navyBlue, periwinkle, pineGreen, seaGreen, greenYellow,
orange, carnationPink, plum

[latex]
preamble =
```

19.3 Setting definitions

We now provide a more detailed description of the effect of each of the settings which can be found in configuration files, including where appropriate a list of possible values for each. Settings are arranged by section.

19.3.1 The `filters` section

The `filters` section allows input filters to be specified for data files whose filenames match particular wildcards. Each line should be in the format

```
wildcard = filter_binary
```

For example, the line

```
*.gz = /usr/bin/gzip
```

would set the application `/usr/bin/gzip` to be used as an input filter for all data files with a `.gz` suffix. For more information about input filters, see Section 5.1.

19.3.2 The `settings` section

The `settings` section can contain any of the following settings in any order:

`aspect`
Possible values: `auto`, or any floating-point number.
Analogous set command: `set size ratio`
Sets the y/x aspect ratio of plots.

`autoAspect`
Possible values: `on`, `off`.
Analogous set command: `set size ratio`
Sets whether plots have the automatic y/x aspect ratio, which is the golden ratio. If `on`, then the `aspect` setting is ignored. Deprecated: new scripts should use `aspect=auto` instead.

`autoZAspect`
Possible values: `on`, `off`.
Analogous set command: `set size zratio`
Sets whether 3d plots have the automatic z/x aspect ratio, which is the golden ratio. If `on`, then the `zAspect` setting is ignored. Deprecated: new scripts should use `zAspect=auto` instead.

`axesColor`
Possible values: Any recognised color.
Analogous set command: `set axescolor`
Sets the color of axis lines and ticks.

`axisUnitStyle`
Possible values: `Bracketed`, `Ratio`, `SquareBracketed`
Analogous set command: `set axisunitstyle`
Sets the style in which the physical units of quantities plotted against axes are appended to axis labels.

`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 tilde after its old filename (see Section 9.4).

`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.

`binOrigin`
Possible values: `auto`, or any floating-point number.
Analogous set command: `set binorigin`
Sets the point along the abscissa axis from which the bins used by the `histogram` command originate.

`binWidth`
Possible values: `auto`, or any floating-point number.
Analogous set command: `set binwidth`
Sets the widths of the bins used by the `histogram` command.

`boxFrom`
Possible values: `auto`, or 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: `auto`, or 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.

`calendarIn`
Possible values: `British`, `French`, `Greek`, `Gregorian`, `Hebrew`, `Islamic`, `Julian`, `Papal`, `Russian`.
Analogous set command: `set calendar`
Sets the default calendar for the input of dates from day, month and year representation into Julian Date representation. See Section 4.11 for more details.

`calendarOut`
Possible values: `British`, `French`, `Greek`, `Gregorian`, `Hebrew`, `Islamic`, `Julian`, `Papal`, `Russian`.
Analogous set command: `set calendar`
Sets the default calendar for the output of dates from Julian Date representation to day, month and year representation. See Section 4.11 for more details.

`clip`
Possible values: `on`, `off`.
Analogous set command: `set clip`
Sets whether datapoints close to the edges of graphs should be clipped at the edges (`on`) or allowed to overrun the axes (`off`).

`colKey`
Possible values: `on`, `off`.
Analogous set command: `set colkey`
Sets whether `colormap` plots have a scale along one side relating color to ordinate value.

`colKeyPos`
Possible values: `top`, `bottom`, `left`, `right`.
Analogous set command: `set colkey`
Sets the side of the plot along which the color legend should appear on `colormap` plots.

`color`
Possible values: `on`, `off`.
Analogous set command: `set terminal`
Sets whether output should be color (`on`) or monochrome (`off`).

`contour`
Possible values: Any integer.
Analogous set command: `set contour`
Sets the number of contours which are drawn in the `contourmap` plot style.

`c?Range_log`
Possible values: `true`, `false`.
Analogous set command: `set logscale c`
When the variables `c1`–`c4` are set to renormalise in the `c?Range_renorm` setting, this setting determines whether color maps are drawn with logarithmic or linear color scales. The `?` wildcard should be replaced with an integer in the range 1–4 to alter the renormalisation of the variables `c1` through `c4` respectively in the expressions supplied to the `colmap` setting. In the case of `c1`, this setting also determines whether contours demark linear or logarithmic intervals on contour maps.

`c?Range_max`
Possible values: Any floating-point number.
Analogous set command: `set crange`
When the variables `c1`–`c4` are set to renormalise in the `c?Range_renorm` setting, this setting determines the upper limit of the range of values demarked by differing colors on color maps. The `?` wildcard should be replaced with an integer in the range 1–4 to alter the renormalisation of the variables `c1` through `c4` respectively in the expressions supplied to the `colmap` setting. In the case of `c1`, this setting also determines the range of ordinate values for which contours are drawn on contour maps.

`c?Range_max_auto`

Possible values: `true`, `false`.
Analogous set command: `set crange`
When the variables `c1`–`c4` are set to renormalise in the `c?Range_renorm` setting, this setting determines whether the upper limit of the range of values demarked by differing colors on color maps should autoscale to fit the data, or be a fixed value as specified in the `C?Range_max` setting. The `?` wildcard should be replaced with an integer in the range 1–4 to alter the renormalisation of the variables `c1` through `c4` respectively. In the case of `c1`, this setting also affects the range of ordinate values for which contours are drawn on contour maps.

`c?Range_min`

Possible values: Any floating-point number.
Analogous set command: `set crange`
When the variables `c1`–`c4` are set to renormalise in the `c?Range_renorm` setting, this setting determines the lower limit of the range of values demarked by differing colors on color maps. The `?` wildcard should be replaced with an integer in the range 1–4 to alter the renormalisation of the variables `c1` through `c4` respectively in the expressions supplied to the `colmap` setting. In the case of `c1`, this setting also determines the range of ordinate values for which contours are drawn on contour maps.

`c?Range_min_auto`

Possible values: `true`, `false`.
Analogous set command: `set crange`
When the variables `c1`–`c4` are set to renormalise in the `c?Range_renorm` setting, this setting determines whether the lower limit of the range of values demarked by differing colors on color maps should autoscale to fit the data, or be a fixed value as specified in the `C?Range_min` setting. The `?` wildcard should be replaced with an integer in the range 1–4 to alter the renormalisation of the variables `c1` through `c4` respectively. In the case of `c1`, this setting also affects the range of ordinate values for which contours are drawn on contour maps.

`c?Range_renorm`

Possible values: `true`, `false`.
Analogous set command: `set crange`
Sets whether the variables `c1`–`c4`, used in the construction of color maps, should be renormalised into the range 0–1 before being passed to the expressions supplied to the `set colmap` command, or whether they should contain the exact data values supplied in the 3rd–6th columns of data to the `colormap` plot style. The `?` wildcard should be replaced with an integer in the range 1–4 to alter the renormalisation of the variables `c1` through `c4` respectively.

`c?Range_reverse`
Possible values: `true`, `false`.
Analogous set command: `set crange`
When the variables `c1`–`c4` are set to renormalise in the `c?Range_renorm` setting, this setting determines whether the renormalisation into the range 0–1 is inverted such that the maximum value maps to zero and the minimum value maps to one. The `?` wildcard should be replaced with an integer in the range 1–4 to alter the renormalisation of the variables `c1` through `c4` respectively.

`dataStyle`
Possible values: Any plot style.
Analogous set command: `set data style`
Sets the plot style used by default when plotting data files.

`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 10.5.3 for more details.

`dpi`
Possible values: Any floating-point number.
Analogous set command: `set terminal dpi`
Sets the sampling quality used, in dots per inch, when output is sent to a bitmapped terminal (the bmp, jpeg, gif, png and tif terminals).

`fontSize`
Possible values: Any floating-point number.
Analogous set command: `set fontsize`
Sets the fontsize of text, where 1.0 represents 10-point text, and other values differ multiplicatively.

`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 horizontal 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 vertical axis to which gridlines should attach, if the `set grid` command is called without specifying which axes to use.

`gridAxisZ`
Possible values: Any integer.
Analogous set command: None
Sets the default z-axis to which gridlines should attach, if the `set grid` command is called without specifying which axes to use.

`gridMajColor`
Possible values: Any recognised color.
Analogous set command: `set gridmajcolor`
Sets the color of major grid lines.

`gridMinColor`
Possible values: Any recognised color.
Analogous set command: `set gridmincolor`
Sets the color 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. If a value of zero is given, then the number of columns is decided automatically for each plot.

`keyPos`
Possible values: `top right`, `top xcenter`, `top left`, `ycenter right`, `ycenter xcenter`, `ycenter left`, `bottom right`, `bottom xcenter`, `bottom left`, `above`, `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.

`numComplex`
Possible values: `on`, `off`.
Analogous set command: `set numerics`
Sets whether complex arithmetic is enabled, or whether all non-real results to calculations should raise numerical exceptions.

`numDisplay`
Possible values: `latex`, `natural`, `typeable`.
Analogous set command: `set numerics`
Sets whether numerical results are displayed in a natural human-readable way, e.g. 2 m, in LaTeX, e.g. `$2\,\mathrm{m}$`, or in a way which may be pasted back into Pyxplot, e.g. `2*unit(m)`.

`numErr`
Possible values: `on`, `off`.
Analogous set command: `set numerics`
Sets whether explicit error messages are thrown when calculations yield undefined results, as in the cases of division by zero or the evaluation of functions in regions where they are undefined or infinite. If explicit error messages are disabled, such calculations quietly return `nan`.

`numSF`
Possible values: Any integer between 0 and 30.
Analogous set command: `set numerics`
Sets the number of significant figures to which numerical quantities are displayed by default.

`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 (case sensitive).
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.

`paperHeight`
Possible values: Any floating-point number.
Analogous set command: `set papersize`
Sets the height of the papersize for PostScript output in millimetres.

`paperName`
Possible values: A string matching any of the papersizes listed in Chapter 16.
Analogous set command: `set papersize`
Sets the papersize for PostScript output to one of the predefined papersizes listed in Chapter 16.

`paperWidth`
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`
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`
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 abscissa axis when plotting a function.

`samples_method`
Possible values: `inverseSquare`, `monaghanLattanzio`, `nearestNeighbor`.
Analogous set command: `set samples`
Sets the method used to interpolate two-dimensional non-gridded arrays of datapoints from datafiles within the `interpolate` command and when plotting using the `colormap`, `contourmap` and `surface` plot styles.

`samples_x`
Possible values: Any integer.
Analogous set command: `set samples`
Sets the number of samples (datapoints) to be evaluated along the first abscissa axis when drawing color maps and surfaces, and when calculating contour maps.

`samples_x_auto`
Possible values: `true`, `false`.
Analogous set command: `set samples`
Sets whether the number of samples (datapoints) to be evaluated along the first abscissa axis when drawing color maps and surfaces, and when calculating contour maps should follow the number of samples set with the `set samples` command.

`samples_y`
Possible values: Any integer.
Analogous set command: `set samples`
Sets the number of samples (datapoints) to be evaluated along the second abscissa axis when drawing color maps and surfaces, and when calculating contour maps.

`samples_y_auto`
Possible values: `true`, `false`.
Analogous set command: `set samples`
Sets whether the number of samples (datapoints) to be evaluated along the second abscissa axis when drawing color maps and surfaces, and when calculating contour maps should follow the number of samples set with the `set samples` command.

`termAntiAlias`
Possible values: `on`, `off`.
Analogous set command: `set terminal`
Sets whether output sent to the bitmapped graphics output terminals – i.e. the bmp, jpeg, gif, png and tif terminals – is antialiased. Antialiasing smooths the color boundaries to disguise the effects of pixelisation and is almost invariably desirable.

`termEnlarge`
Possible values: `on`, `off`.
Analogous set command: `set terminal`
When set to `on` output is enlarged or shrunk to fit the current paper size.

`termInvert`
Possible values: `on`, `off`.
Analogous set command: `set terminal`
Sets whether jpeg/gif/png output has normal colors (`off`), or inverted colors (`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: `bmp`, `eps`, `gif`, `jpg`, `pdf`, `png`, `ps`, `svg`, `tif`, `X11_multiWindow`, `X11_persist`, `X11_singleWindow`.
Analogous set command: `set terminal`
Sets whether output is sent to the screen, using one of the `X11_...` terminals, or to disk. In the latter case, output may be produced in a wide variety of graphical formats.

`textColor`
Possible values: Any recognised color.
Analogous set command: `set textcolor`
Sets the color of all text output.

`textHAlign`
Possible values: `left`, `center`, `right`.
Analogous set command: `set texthalign`
Sets the horizontal alignment of text labels to their given reference positions.

`textVAlign`
Possible values: `top`, `center`, `bottom`.
Analogous set command: `set textvalign`
Sets the vertical alignment of text labels to their given reference positions.

`title`
Possible values: Any string (case sensitive).
Analogous set command: `set title`
Sets the title to appear at the top of the plot.

`title_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.

`title_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.

`tRange_log`
Possible values: `true`, `false`.
Analogous set command: `set logscale t`
Sets whether the t-axis – used for parametric plotting – is linear or logarithmic.

`tRange_max`
Possible values: Any floating-point number.
Analogous set command: `set trange`
Sets upper limit of the t-axis, used for parametric plotting.

`tRange_min`
Possible values: Any floating-point number.
Analogous set command: `set trange`
Sets lower limit of the t-axis, used for parametric plotting.

`unitAbbrev`
Possible values: `on`, `off`.
Analogous set command: `set unit`
Sets whether physical units are displayed in abbreviated form, e.g. `mm`, or in full, e.g. `millimetres`.

`unitAngleDimless`
Possible values: `on`, `off`.
Analogous set command: `set unit`
Sets whether angles are treated as dimensionless units, or whether the radian is treated as a base unit.

`unitPrefix`
Possible values: `on`, `off`.
Analogous set command: `set unit`
Sets whether SI prefixes, such as `milli-` and `mega-` are prepended to SI units where appropriate.

`unitScheme`
Possible values: `ancient`, `cgs`, `imperial`, `planck`, `si`, `USCustomary`.
Analogous set command: `set unit`
Sets the scheme of physical units in which quantities are displayed.

`uRange_log`
Possible values: `true`, `false`.
Analogous set command: `set logscale u`
Sets whether the u-axis – used for parametric plotting – is linear or logarithmic.

`uRange_max`
Possible values: Any floating-point number.
Analogous set command: `set urange`
Sets upper limit of the u-axis, used for parametric plotting.

`uRange_min`
Possible values: Any floating-point number.
Analogous set command: `set urange`
Sets lower limit of the t-axis, used for parametric plotting.

`vRange_log`
Possible values: `true`, `false`.
Analogous set command: `set logscale v`
Sets whether the v-axis – used for parametric plotting – is linear or logarithmic.

`vRange_max`
Possible values: Any floating-point number.
Analogous set command: `set vrange`
Sets upper limit of the v-axis, used for parametric plotting.

`vRange_min`
Possible values: Any floating-point number.
Analogous set command: `set vrange`
Sets lower limit of the v-axis, used for parametric plotting.

`width`
Possible values: Any floating-point number.
Analogous set commands: `set width`, `set size`
Sets the width of plots in centimetres.

`view_xy` **Possible values:** Any floating-point number.
Analogous set commands: `set view`
Sets the viewing angle of three-dimensional plots in the x-y plane in degrees.

`view_yz` **Possible values:** Any floating-point number.
Analogous set commands: `set view`
Sets the viewing angle of three-dimensional plots in the y-z plane in degrees.

`zAspect` **Possible values:** `auto`, or any floating-point number.
Analogous set command: `set size ratio`
Sets the z/x aspect ratio of 3d plots.

19.3.3 The `styling` section

The `styling` section can contain any of the following settings in any order:

`arrow_headAngle` **Possible values:** Any floating-point number.
Sets the angle, in degrees, at which the two sides of arrow heads meet at its point.

`arrow_headSize` **Possible values:** Any floating-point number.
Sets the size of all arrow heads. A value of 1.0 corresponds to Pyxplot's default size.

`arrow_headBackIndent` **Possible values:** Any floating-point number.
Sets the size of the indentation in the back of arrow heads. The default size is 0.2. Sensible values lie in the range 0 (no indentation) to 1 (the indentation extends the whole length of the arrow head). Less sensible values may be used by the aesthetically adventurous.

`axes_lineWidth` **Possible values:** Any floating-point number.
Sets the line width used to draw graph axes.

`axes_majTickLen` **Possible values:** Any floating-point number.
Sets the length of major axis ticks. A value of 1.0 corresponds to Pyxplot's default length of 1.2 mm; other values differ from this multiplicatively.

`axes_minTickLen` **Possible values:** Any floating-point number.
Sets the length of minor axis ticks. A value of 1.0 corresponds to Pyxplot's default length of 0.85 mm; other values differ from this multiplicatively.

`axes_separation` **Possible values:** Any floating-point number.
Sets the separation between parallel axes on graphs, less the width of any text labels associated with the axes. A value of 1.0 corresponds to Pyxplot's default spacing of 8 mm; other values differ from this multiplicatively.

`axes_textGap` **Possible values:** Any floating-point number.
Sets the separation between axes and the text labels which are associated with them. A value of 1.0 corresponds to Pyxplot's default spacing of 3 mm; other values differ from this multiplicatively.

`colorScale_margin` **Possible values:** Any floating-point number.
Sets the separation left between the axes of plots drawn using the `colormap` plot style, and of the color scales drawn alongside them. A value of 1.0 corresponds to Pyxplot's default spacing; other values differ from this multiplicatively.

`colorScale_width` **Possible values:** Any floating-point number.
Sets the width of the color scale bars drawn alonside plots drawn using the `colormap` plot style. A value of 1.0 corresponds to Pyxplot's width; other values differ from this multiplicatively.

`grid_majLineWidth` **Possible values:** Any floating-point number.
Sets the line width used to draw major gridlines (default 1.0).

`grid_minLineWidth` **Possible values:** Any floating-point number.
Sets the line width used to draw minor gridlines (default 0.5).

`baseline_lineWidth` **Possible values:** Any floating-point number.
Sets the PostScript line width which corresponds to a `linewidth` of 1.0. A value of 1.0 corresponds to Pyxplot's default line width of 0.2 mm; other values differ from this multiplicatively.

`baseline_pointSize` **Possible values:** Any floating-point number.
Sets the baseline point size which corresponds to a `pointsize` of 1.0. A value of 1.0 corresponds to Pyxplot's default; other values differ from this multiplicatively.

19.3.4 The `terminal` section

The `terminal` section can contain any of the following settings in any order:

`color`
Possible values: `on`, `off`.
Analogous command-line switches: `-c`, `--color`, `-m`, `--monochrome`.
Sets whether color 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 colors are configurable, as described below. Note that not all UNIX terminals support the use of color.

`color_err`
Possible values: Any recognised terminal color (see below).
Analogous command-line switches: None.
Sets the color in which error messages are displayed when color highlighting is used. Note that the list of recognised color names differs from that used in Pyxplot; a list is given at the end of this section.

`color_rep`
Possible values: Any recognised terminal color (see below).
Analogous command-line switches: None.
As above, but sets the color in which Pyxplot displays its non-error-related output.

`color_wrn`
Possible values: Any recognised terminal color (see below).
Analogous command-line switches: None.
As above, but sets the color 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 colors recognised by the `COLOR_XXX` configuration options above are: `Red`, `Green`, `Amber`, `Blue`, `Purple`, `Magenta`, `Cyan`, `White`, `Normal`. The final option produces the default foreground color of the terminal.

19.3.5 The `units` section

The `units` section can be used to define new physical units for use within Pyxplot's mathematical environment. Each line should take the format of

```
<l_sing> \[ / <s_sing> \] \[ / <lt_sing> \]
   \[ / <l_plur> \] \[ / <s_plur> \] \[ / <lt_plur> \]
   : <quantity_name> = \[ <definition> \]
```

where

`l_sing`	is the long singular name of the unit, e.g. `metre`.
`s_sing`	is the short singular name of the unit, e.g. `m`.
`lt_sing`	is the singular name of the unit to be used in latex.
`l_plur`	is the long plural name of the unit, e.g. `metres`.
`s_plur`	is the short plural name of the unit, e.g. `m`.
`lt_plur`	is the plural name of the unit to be used in latex.
`quantity_name`	is the physical quantity which the unit measures, e.g. `length`.
`definition`	is a definition of the unit in terms of other units which Pyxplot already recognises, e.g. `0.001*km`. The syntax used is identical to that used in the `unit()` function.

For example, a definition of the metre would look like

```
metre/m/m/metres/m/m:length=0.001*km
```

Not all of the various names which a unit may have need to be specified. If plural names are not specified then they are assumed to be the same as the singular names. If short and/or latex names are not specified they are assumed to be the same as the long name. If the definition is left blank then the unit is assumed to be a new base unit which is not related to any pre-existing units.

19.4 Recognised color names

The following is a complete list of the color names which Pyxplot recognises in the `set textcolor`, `set axescolor` commands, and in the `colors` section of its configuration file. A color chart of these can be found in Appendix 17. All color names 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.
```

In addition, a scale of 100 shades of grey is available, ranging from `grey00`, which is black, to `grey99`, which is very nearly white. The US spelling, `gray??`, is also accepted.

Arbitrary colors may be specified in the forms `rgb0:0:0`, `hsb0:0:0` or `cmyk0:0:0:0`, where the colon-separated zeros should be replaced by values in the range of 0 to 1 to represent the components of the desired color in RGB, HSB or CMYK space respectively.

Part IV

Appendices

Appendix A

Other applications of Pyxplot

In this chapter we present a short cookbook describing a few common yet miscellaneous tasks for which Pyxplot may prove useful.

A.1 Conversion of jpeg images to PostScript

The need to convert bitmap images – for example, those in jpeg format – into PostScript representations is commonly encountered by users of the latex typesetting system, since latex's `includegraphics` command can only incorporate Encapsulated PostScript (EPS) images into documents. A small number of graphics packages provide facilities for making such conversions, including ImageMagick's `convert` command, but these almost invariable produce excessively large PostScript files on account of their failure to use PostScript's native facilities for image compression. Pyxplot's `image` command can in many cases perform much more efficient conversion:

```
set output image.eps
image 'image.jpg' width 10
```

A.2 Inserting equations in Powerpoint presentations

The two tools most commonly used for presenting talks – Microsoft *Powerpoint* and OpenOffice *Impress* – have limited facilities for importing text rendered in latex into slides. *Powerpoint* does include its own *Equation Editor*, but its output is considerably less professional than that produced by latex. This can prove a frustration for anyone who works in a field with notation which makes use of non-standard characters, but especially for those who work in mathematical and equation-centric disciplines.

It is possible to import graphic images into *Powerpoint*, but it cannot read images in PostScript format, the format in which latex usually produces its

output. Pyxplot's `gif` and `png` terminals provide a fix for this problem, as the following example demonstrates:

```
set term transparent noantialias gif
set term dpi 300
set output 'equation.gif'
set multiplot

# Render the Planck blackbody formula in LaTeX
set textcolour yellow
text '$B_\nu = \frac{8\pi h}{c^3} \
\frac{\nu^3}{\exp \left( h\nu / kT \right) -1 }$' at 0,0
text 'The Planck Blackbody Formula:' at 0 , 0.75
```

The result is a `gif` image of the desired equation, with yellow text on a transparent background. This can readily be imported into *Powerpoint* and re-scaled to the desired size.

A.3 Delivering talks in Pyxplot

Going one step further, Pyxplot can be used as a stand-alone tool for designing slides for talks; it has several advantages over other presentation tools. All of the text which is placed on slides is rendered neatly in latex. Images can be placed on slides using the `jpeg` and `eps` commands, and placed at any arbitrary coordinate position on the slide. In comparison with programs such as Microsoft *Powerpoint* and OpenOffice *Impress*, the text looks much neater, especially if equations or unusual characters are required. In comparison with TEX-based programs such as FoilTEX, it is much easier to incorporate images around text to create colourful slides which will keep an audience attentive.

As an additional advantage, graphs can be plotted within the scripts describing each slide, directly from data files in your local filesystem. If you receive new data shortly before giving a talk, it is a simple matter to re-run the Pyxplot scripts and your slides will automatically pick up the new data files.

Below, we outline our recipe for designing slides in Pyxplot. There are many steps, but they do not take much time; many simply involve pasting text into various files. Readers of the printed version of the manual may find it easier to copy these files from the HTML version of this manual on the Pyxplot website.

A.3.1 Setting up the infrastructure

First, a bit of infrastructure needs to be set up. Note that once this has been done for one talk, the infrastructure can be copied directly from a previous talk.

1. Make a new directory in which to put your talk:

   ```
   mkdir my_talk
   cd my_talk
   ```

2. Make a directory into which you will put the Pyxplot scripts for your individual slides:

```
mkdir scripts
```

3. Make a directory into which you will put any graphic images which you want to put into your talk to make it look pretty:

   ```
   mkdir images
   ```

4. Make a directory into which Pyxplot will put graphic images of your slides:

   ```
   mkdir slides
   ```

5. Design a background for your slides. Open a paint program such as the `gimp`, create a new image which measures 1024×768 pixels, and fill it with colour. My preference tends to be for a blue colour gradient, running from bright blue at the top to dark blue at the bottom, but you may be more inventive than me. You may wish to add institutional and/or project logos in the corners. Alternatively, you can download a ready-made background image from the Pyxplot website: http://foo. You should store this image as `images/background.jpg`.

6. We need a simple Pyxplot script to set up a slide template. Paste the following text into the file `scripts/slide_init`; there's a bit of black magic in the `arrow` commands in this script which it isn't necessary to understand at this stage:

   ```
   scale  = 1.25        ; inch = 2.54 # cm
   width  = 10.24*scale ; height =  7.68*scale
   x = width/100.0      ; y = height/100.0
   set term gif ; set dpi (1024.0/width) * inch
   set multiplot ; set nodisplay
   set texthalign centre ; set textvalign centre
   set textcolour yellow
   jpeg "images/background.jpg" width width
   arrow -x* 25,-y* 25 to -x* 25, y*125 with nohead
   arrow -x* 25, y*125 to  x*125, y*125 with nohead
   arrow  x*125, y*125 to  x*125,-y* 25 with nohead
   arrow  x*125,-y* 25 to -x* 25,-y* 25 with nohead
   ```

7. We also need a simple Pyxplot script to round off each slide. Paste the following text into the file `scripts/slide_finish`:

   ```
   set display ; refresh
   ```

8. Paste the following text into the file `compile`. This is a simple shell script which instructs `pyxplot_watch` to compile your slides using Pyxplot every time you edit any of the them:

   ```
   #!/bin/bash
   pyxplot_watch --verbose scripts/0\*
   ```

9. Paste the following text into the file `make_slides`. This is a simple shell script which crops your slides to measure exactly 1024×768 pixels, cropping any text boxes which may go off the side of them. It links up with the black magic of Step 6:

```
#!/bin/bash
mkdir -p slides_cropped
for all in slides/*.gif ; do
convert $all -crop 1024x768+261+198 `echo $all | \
sed 's@slides@slides_cropped@' | sed 's@gif@jpg@'`
done
```

10. Make the scripts `compile` and `make_slides` executable:

```
chmod 755 compile make_slides
```

A.3.2 Writing a short example talk

The infrastructure is now completely set up, and you are ready to start designing slides. We will now design an example talk with three slides.

1. Run the script `compile` and leave it running in the background. Pyxplot will then re-run the scripts describing your slides whenever you edit them.

2. As an example, we will now make a title slide. Paste the following script into the file `scripts/0001`:

```
set output 'slides/0001.gif'
load 'scripts/slide_init'

text '\parbox[t]{10cm}{\center \LARGE \bf \
  A Tutorial in the use of Pyxplot \\ \
  to present Talks \
} ' at x*50, y*75
text '\Large \bf Prof A.N.\ Other' at x*50, y*45
text '\parbox[t]{9cm}{\center \
  Director, \\ \
  Atlantis Island University \
} ' at x*50, y*38
text 'Annual Lecture, 1st January 2010' at x*50, y*22

load 'scripts/slide_finish'
```

   Note that the variables `x` and `y` are defined to be 1 per cent of the width and height of your slides respectively, such that the bottom-left of each slide is at $(0, 0)$ and the top-right of each slide is at $(100 * \mathtt{x}, 100 * \mathtt{y})$.

3. Next we will make a second slide with a series of bullet points. Paste the following script into the file `scripts/0002`:

```
set output 'slides/0002.gif'
load 'scripts/slide_init'

text '\Large \textbf{Talk Overview}' at x*50, y*92
text "\parbox[t]{9cm}{\begin{itemize} \
 \item Setting up the Infrastructure. \
 \item Writing a Short Example Talk. \
 \item Delivering your Talk. \
 \item Conclusion. \
 \end{itemize} \
} " at x*50 , y*60

set textcol cyan
text '{\bf With thanks to my collaborator, \
           Prof Y.E.\ Tanother.}' at x*50,y*15

load 'scripts/slide_finish'
```

4. Finally, we will make a third slide with a graph on it. Paste the following script into the file `scripts/0003`:

```
set output 'slides/0003.gif'
load 'scripts/slide_init'

text '\Large \bf The Results of Our Model' at x*50, y*92
set axescolour yellow ; set nogrid
set origin x*17.5, y*20 ; set width x*70
set xrange [0.01:0.7]
set xlabel '$x$'
set yrange [0.01:0.7]
set ylabel '$f(x)$'
set palette Red, Green, Orange, Purple

set key top left
plot x t 'Model 1', exp(x)-1 t 'Model 2', \
     log(x+1) t 'Model 3', sin(x) t 'Model 4'

load 'scripts/slide_finish'
```

5. To view your slides, run the script `make_slides`. Afterwards, you will find your slides as a series of 1024×768 pixel jpeg images in the directory `slides_cropped`. If you have the *Quick Image Viewer* (`qiv`) installed, then you can view them as follows:

   ```
   qiv slides_cropped/*
   ```

   If you're in a hurry, you can skip the step of running the script `make_slides` and view your slides as images in the `slides` directory, but note that the slides in here may not be properly cropped. This approach is generally preferable when viewing your slides in a semi-live fashion as you are editing them.

6. If you'd like to make the text on your slides larger or smaller, you can do so by varying the `scale` parameter in the file `scripts/slide_init`.

A.3.3 Delivering your talk

There are two straightforward ways in which you can give your talk. The quickest way is simply to use the *Quick Image Viewer* (`qiv`):

```
qiv slides_cropped/*
```

Press the left mouse button to move forward through your talk, and the right mouse button to go back a slide.

This method does lack some of the niceties of Microsoft *Powerpoint* – for example, the ability to jump to any arbitrary slide number, compatibility with wireless remote controls to advance your slides, and the ability to use animated slide transitions. It may be preferably, therefore, to paste the jpeg images of your slides into a *Powerpoint* or OpenOffice *Impress* presentation before you give your talk.

Appendix B

Summary of differences between Pyxplot and gnuplot

Pyxplot's command-line interface is based loosely upon that of gnuplot, but does not completely re-implement the entirety of gnuplot's command language. Moreover, Pyxplot's command language includes many extensions of gnuplot's interface. In this Appendix, we outline some of the most significant areas in which gnuplot and Pyxplot differ. This is far from an exhaustive list, but may provide a useful reference for gnuplot users.

B.1 The typesetting of text

Pyxplot renders all text labels automatically in the latex typesetting environment. This brings many advantages: it produces neater labels than the default typesetting engine used by gnuplot, makes it straightforward to label graphs with mathematical expressions, and moreover makes it straightforward when importing graphs into latex documents to match the fonts used in figures with those used in the main text of the document. It does, however, also necessarily introduce some incompatibility with gnuplot. Some strings which are valid in gnuplot are not valid in Pyxplot (see Section 3.6 for more details). For example,

✗
```
set xlabel 'x^2'
```

is a valid label in gnuplot, but is not valid input for latex and therefore fails in Pyxplot. In Pyxplot, it needs to be written in latex mathmode as:

✔
```
set xlabel '$x^2$'
```

A useful introduction to latex's syntax can be found in Tobias Oetiker's excellent free tutorial, *The Not So Short Guide to latex* 2ϵ, which is available for free download from:
http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf

Two built-in functions provide some assistance in generating latex labels. The `texify()` takes as its argument a string containing a mathematical expression, and returns a latex representation of it. The `texifyText()` takes as its argument a text string, and returns a latex representation of it, with any necessary escape characters added. For example:

```
pyxplot> print texify("sqrt(x**2+1)")
$\displaystyle \sqrt{x\,*\kern-1.5pt *\,\,2+1}\mathrm{}$
pyxplot> a=50
pyxplot> print texifyText("A %d%% increase"%a)
A 50\% increase
pyxplot> set ylabel texify("cos(x**2)")
```

Two built-in functions provide some assistance in generating latex labels. The `texify()` takes as its argument a string containing a mathematical expression, and returns a latex representation of it. The `texifyText()` takes as its argument a text string, and returns a latex representation of it, with any necessary escape characters added. For example:

```
pyxplot> print texify("sqrt(x**2+1)")
$\displaystyle \sqrt{x\,*\kern-1.5pt *\,\,2+1}\mathrm{}$
pyxplot> a=50
pyxplot> print texifyText("A %d%% increase"%a)
A 50\% increase
pyxplot> set ylabel texify("cos(x**2)")
```

B.2 Complex numbers

The syntax used for representing complex numbers in Pyxplot differs from that used in gnuplot. Whereas gnuplot expects the real and imaginary components of complex numbers to be represented {a,b}, Pyxplot uses the syntax `a+b*i`, assuming that the variable `i` has been defined to equal `sqrt(-1)`. In addition, in Pyxplot complex arithmetic must first be enabled using the `set numerics complex` command before complex numbers may be entered. This is illustrated by the following example:

```
gnuplot> print {1,2} + {3,4}
{4.0, 6.0}

pyxplot> set numerics complex
pyxplot> print (1+2*i) + (3+4*i)
(4+6i)
```

B.3 The multiplot environment

gnuplot's multiplot environment, used for placing many graphs alongside one another, is massively extended in Pyxplot. As well as making it much easier to produce galleries of plots and inset graphs, a wide range of vector graphs

objects can also be added to the multiplot canvas. This is described in detail in Chapter 10.

B.4 Plots with multiple axes

In gnuplot, a maximum of two horizontal and two vertical axes may be associated with each graph, placed in each case with one on either side of the plot. These are referred to as the `x` (bottom) and `x2` (top), or `y` (left) and `y2` (right) axes. This behaviour is reproduced in Pyxplot, and so the syntax

```
set x2label 'Axis label'
```

works similarly in both programs. However, in Pyxplot the position of each axis may be set individually using syntax such as

```
set axis x2 top
```

and furthermore up to 128 axes may be placed parallel to one another:

```
set axis x127 top
set x127label "This is axis number 127"
```

More details of how to configure axes can be found in Section 8.8.1.

B.5 Plotting parametric functions

The syntax used for plotting parametric functions differs between gnuplot and Pyxplot. Whereas parametric plotting is enabled in gnuplot using the `set parametric` command, in Pyxplot it is enabled on a per-dataset basis by placing the keyword `parametric` before the algebraic expression to be plotted:

```
gnuplot> set parametric
gnuplot> set trange [0:2*pi]
gnuplot> plot sin(t),cos(t)
```

```
pyxplot> set trange [0:2*pi]
pyxplot> plot parametric sin(t):cos(t)
```

This makes it straightforward to plot parametric functions alongside non-parametric functions. For more information, see Section 8.6.

Appendix C

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*.

C.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 data file 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 data file 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 $\mathrm{P}\left(\mathbf{u} | \left\{\mathbf{x}_i, f_i, \sigma_i\right\}\right)$.

C.2 The probability density function

Bayes' Theorem states that:

$$\mathrm{P}\left(\mathbf{u}|\left\{\mathbf{x}_i, f_i, \sigma_i\right\}\right) = \frac{\mathrm{P}\left(\left\{f_i\right\}|\mathbf{u}, \left\{\mathbf{x}_i, \sigma_i\right\}\right)\mathrm{P}\left(\mathbf{u}|\left\{\mathbf{x}_i, \sigma_i\right\}\right)}{\mathrm{P}\left(\left\{f_i\right\}|\left\{\mathbf{x}_i, \sigma_i\right\}\right)} \tag{C.1}$$

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 $\mathrm{P}\left(\mathbf{u}\right)$ 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:

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

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:

$$L = \sum_{i=0}^{n_\mathrm{d}-1} \left(\frac{-\left[f_i - f_\mathbf{u}(\mathbf{x}_i)\right]^2}{2\sigma_i^2}\right) + k \tag{C.3}$$

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 $\mathbf{u}^0$.

C.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:

$$\begin{aligned} L(\mathbf{u}) \quad = \quad & L(\mathbf{u}^0) + \underbrace{\sum_{i=0}^{n_\mathrm{u}-1} \left(u_i - u_i^0\right) \left.\frac{\partial L}{\partial u_i}\right|_{\mathbf{u}^0}}_{\text{Zero at } \mathbf{u}^0 \text{ by definition}} + \\ & \sum_{i=0}^{n_\mathrm{u}-1}\sum_{j=0}^{n_\mathrm{u}-1} \frac{\left(u_i - u_i^0\right)\left(u_j - u_j^0\right)}{2} \left.\frac{\partial^2 L}{\partial u_i \partial u_j}\right|_{\mathbf{u}^0} + \mathcal{O}\left(\mathbf{u} - \mathbf{u}^0\right)^3 \end{aligned} \tag{C.4}$$

Since the logarithm of a Gaussian distribution is a parabola, the quadratic terms in the above expansion encode the Gaussian component of the probability

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

$$Q = \frac{1}{2}\left(\mathbf{u}-\mathbf{u}^0\right)^{\mathbf{T}}\mathbf{A}\left(\mathbf{u}-\mathbf{u}^0\right) \tag{C.5}$$

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

$$A_{ij} = \nabla\nabla L = \left.\frac{\partial^2 L}{\partial u_i \partial u_j}\right|_{\mathbf{u}^0} \tag{C.6}$$

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 (C.4) yields elliptical contours of equi-probability 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 (C.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 equi-probability contours become the equations of ellipses:

$$Q = \frac{1}{2}\sum_{i=0}^{n_{\mathrm{u}}-1} A_{ii}\left(u_i - u_i^0\right)^2 = k \tag{C.7}$$

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.

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 equi-probability 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.

C.4 The covariance matrix

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

$$V_{ij} = \left\langle\left(u_i - u_i^0\right)\left(u_j - u_j^0\right)\right\rangle \tag{C.8}$$

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.

[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).

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 C.5.

Using Δu_i to denote $\left(u_i - u_i^0\right)$, we may proceed by rewriting Equation (C.8) as:

$$\begin{aligned} V_{ij} &= \int \cdots \int_{u_i=-\infty}^{\infty} \Delta u_i \Delta u_j \mathrm{P}\left(\mathbf{u}|\left\{\mathbf{x}_i, f_i, \sigma_i\right\}\right) \mathrm{d}^{n_\mathrm{u}}\mathbf{u} \\ &= \frac{\int \cdots \int_{u_i=-\infty}^{\infty} \Delta u_i \Delta u_j \exp(-Q)\, \mathrm{d}^{n_\mathrm{u}}\mathbf{u}}{\int \cdots \int_{u_i=-\infty}^{\infty} \exp(-Q)\, \mathrm{d}^{n_\mathrm{u}}\mathbf{u}} \end{aligned} \tag{C.9}$$

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:

$$\begin{aligned} Z &= \int \cdots \int_{u_i=-\infty}^{\infty} \exp\left(\frac{1}{2}\Delta\mathbf{u}^\mathbf{T}\mathbf{A}\Delta\mathbf{u}\right) \mathrm{d}^{n_\mathrm{u}}\mathbf{u} \\ &= \frac{(2\pi)^{n_\mathrm{u}/2}}{\mathrm{Det}(-\mathbf{A})} \end{aligned} \tag{C.10}$$

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

$$-2\frac{\partial}{\partial A_{ij}}\left[\log_e(Z)\right] = \frac{1}{Z}\int \cdots \int_{u_i=-\infty}^{\infty} \Delta u_i \Delta u_j \exp(-Q)\, \mathrm{d}^{n_\mathrm{u}}\mathbf{u} \tag{C.11}$$

which we may identify as equalling V_{ij}:

$$\begin{aligned} V_{ij} &= -2\frac{\partial}{\partial A_{ij}}\left[\log_e(Z)\right] \\ &= -2\frac{\partial}{\partial A_{ij}}\left[\log_e((2\pi)^{n_\mathrm{u}/2}) - \log_e(\mathrm{Det}(-\mathbf{A}))\right] \\ &= 2\frac{\partial}{\partial A_{ij}}\left[\log_e(\mathrm{Det}(-\mathbf{A}))\right] \end{aligned} \tag{C.12}$$

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

$$\frac{\partial}{\partial A_{ij}}\left[\mathrm{Det}(-\mathbf{A})\right] = -a_{ij} \tag{C.13}$$

where a_{ij} is the cofactor of A_{ij}. Substituting this into Equation (C.12) yields:

$$V_{ij} = \frac{-a_{ij}}{\mathrm{Det}(-\mathbf{A})} \tag{C.14}$$

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

$$V_{ij} = \frac{-\mathbf{A}^{\dagger}}{\mathrm{Det}(-\mathbf{A})} \equiv (-\mathbf{A})^{-1} \tag{C.15}$$

which proves the result stated earlier.

C.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 C.6.

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

$$C_{ij} = \frac{V_{ij}}{\sigma_i \sigma_j} \tag{C.16}$$

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.

C.6 Finding σ_i

Throughout the preceding sections, the uncertainties in the supplied target values f_i have been denoted σ_i (see Section C.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 C.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:

$$\mathrm{P}\left(\sigma_{\mathrm{data}} | \left\{\mathbf{x}_i, f_i\right\}\right) = \frac{\mathrm{P}\left(\left\{f_i\right\} | \sigma_{\mathrm{data}}, \left\{\mathbf{x}_i\right\}\right) \mathrm{P}\left(\sigma_{\mathrm{data}} | \left\{\mathbf{x}_i\right\}\right)}{\mathrm{P}\left(\left\{f_i\right\} | \left\{\mathbf{x}_i\right\}\right)} \tag{C.17}$$

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 $\mathbf{u}$:

$$\mathrm{P}\left(\{f_i\}\,|\sigma_{\mathrm{data}},\{\mathbf{x}_i\}\right)=\int\cdots\int_{-\infty}^{\infty}\mathrm{P}\left(\{f_i\}\,|\sigma_{\mathrm{data}},\{\mathbf{x}_i\},\mathbf{u}\right)\times \mathrm{P}\left(\mathbf{u}|\sigma_{\mathrm{data}},\{\mathbf{x}_i\}\right)\,\mathrm{d}^{n_{\mathrm{u}}}\mathbf{u} \tag{C.18}$$

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

$$\int\cdots\int_{-\infty}^{\infty}\exp\left(\frac{1}{2}\mathbf{u}^{\mathbf{T}}\mathbf{A}\mathbf{u}\right)\,\mathrm{d}^{n_{\mathrm{u}}}\mathbf{u}=\frac{(2\pi)^{n_{\mathrm{u}}/2}}{\sqrt{\mathrm{Det}\left(-\mathbf{A}\right)}} \tag{C.19}$$

We may thus approximate Equation (C.18) as:

$$\mathrm{P}\left(\{f_i\}\,|\sigma_{\mathrm{data}},\{\mathbf{x}_i\}\right)\approx\mathrm{P}\left(\{f_i\}\,|\sigma_{\mathrm{data}},\{\mathbf{x}_i\},\mathbf{u}^0\right)\times \mathrm{P}\left(\mathbf{u}^0|\sigma_{\mathrm{data}},\{\mathbf{x}_i,f_i\}\right)\frac{(2\pi)^{n_{\mathrm{u}}/2}}{\sqrt{\mathrm{Det}\left(-\mathbf{A}\right)}} \tag{C.20}$$

As in Section C.2, it is numerically easier to maximise this quantity via its logarithm, which we denote L_2, and can write as:

$$L_2=\sum_{i=0}^{n_{\mathrm{d}}-1}\left(\frac{-\left[f_i-f_{\mathbf{u}^0}(\mathbf{x}_i)\right]^2}{2\sigma_{\mathrm{data}}^2}-\log_e\left(2\pi\sqrt{\sigma_{\mathrm{data}}}\right)\right)+\log_e\left(\frac{(2\pi)^{n_{\mathrm{u}}/2}}{\sqrt{\mathrm{Det}\left(-\mathbf{A}\right)}}\right) \tag{C.21}$$

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

Appendix D

ChangeLog

2012 Sep 19: Pyxplot 0.9.2

Version 0.9.2 corrects a large number of minor bugs.

2012 Aug 29: Pyxplot 0.9.1

Version 0.9.1 is a minor update with new support for running Pyxplot on Raspberry Pi. It fixes SIGBUS errors in Pyxplot's math engine when run on armhf architectures.

2012 Aug 1: Pyxplot 0.9.0

Version 0.9 is a major update. Many new data types have been introduced, each of which has methods which can be called in an object-orientated fashion. These include:

- **Colors**, which can be stored in variables for subsequent use in vector graphics commands. The addition and subtraction operators act on colors to allow color mixing.
- **Dates**, which can be imported from calendar dates, unix times or Julian dates. Dates can be subtracted to give time intervals.
- **Lists** and **dictionaries**, which can be iterated over, or used to feed calculated data into the `plot` and `tabulate` commands.
- **Vectors** and **matrices**, which allow matrix algebra. These types interface cleanly with Pyxplot's vector-graphics commands, allowing positions to be specified as vector expressions.
- **File handles**, which allow Pyxplot to read data from files, or write data or logs to files.
- **Modules** and **classes**, which allow object-orientated programming.

In addition, Pyxplot's range of operators has been extended to include most of those in the C programming language, allowing expressions such as

```
pyxplot> print (a=3)+(b=2)
5
pyxplot> print a>0?"yes":"no"
yes
pyxplot> print "%s  %s"%(++a,b++)
4  2
pyxplot> print (a+=10 , b+=10 , a+b)
27
```

to be written.

Incompatibilities with Pyxplot 0.8

The extensions to Pyxplot in version 0.9 mean that some minor changes to syntax have been necessary. These include:

- Some functions and variables have been renamed. Variables whose names used to begin `phy_` now live in a module called `phy`. They may be accessed as, for example, `phy.c`. Similarly, random number generating functions now live in a module called `random`; statistics functions in a module called `stats`; time-handling functions in `time`; operating system functions in `os`; and astronomy functions in `ast`. The contents of these modules can be listed by typing, for example, `print phy`.
- Custom colors, which used to be specified using syntax such as `rgb0.2:0.3:0.4`, should now be specified using the `rgb(r,g,b)` functions, as, for example, `rgb(0.2,0.3,0.4)`. Custom colors can now be stored in variables for later use (see Section 6.6).
- The range of escape characters which can be used in strings has been increased, so that, for example, `\n` is a newline and `\t` a tab. As in python, prepending the string with the character `r` disables all escape character expansion. As backslashes are common characters in latex command strings, the easiest approach is to always prepend latex strings with an `r`. As in python, triple quotes, e.g. `r"""2 \times 3"""` can be used where required (see Section 3.6).
- In the `foreach` command, square brackets should be used to delimit lists of items to iterate over. The Pyxplot 0.8 syntax `foreach i in (1,2,3)` should now be written `foreach i in [1,2,3]` (see Section 7.3).

2011 Jan 7: Pyxplot 0.8.4

Summary:

This is a minor bugfix release.

Details:

- Two-dimensional parametric grid plotting implemented.
- Bugfix to the dots plot style; filled triangles replaces with filled circles.

- Bugfix to linewidths used when drawing line icons on graph legends.
- Bugfix to Makefile to ensure libraries link correctly under Red Hat and SUSE.
- Code cleanup to ensure correct compilation with `-O2` optimisation.

2010 Sep 15: Pyxplot 0.8.3

Summary:

This is a minor bugfix release.

Details:

- @ macro expansion operator implemented.
- assert command implemented.
- for command behaviour changed such that `for i=1 to 10` includes a final iteration with `i`=10.
- Point types rearranged into a more logical order.
- Improved support for newer Windows bitmap images.
- Bugfix to the `set unit preferred` command.
- Binary not operator bugfixed.
- Bugfix to handling of comma-separated horizontal datafiles.
- Mathematical function `finite()` added.

2010 Aug 4: Pyxplot 0.8.2

Summary:

This release introduces three-dimensional plotting, as well as the ability to plot two-dimensional maps of functions as either color maps, contour plots, or as three-dimensional surfaces. A large number of bugs have also been fixed.

Details:

- 3D plotting implemented.
- New plot styles colormap, contourmap and surface added.
- Interpolation of 2D datagrids and bitmap images implemented.
- Stepwise interpolation mode added.
- Dependency on libkpathsea relaxed to make installation under MacOS easier; linking to the library is still strongly recommended on systems where it is readily available.
- Mathematical functions `frac-tal_julia()`, `fractal_mandelbrot()` and `prime()` added.
- Many bug fixes, especially to the ticking of axes.

2010 Jun 1: Pyxplot 0.8.1

Summary:

This release has no major new features, but fixes several significant bugs in version 0.8.0.

Details:

- Mathematical functions `time_fromunix()`, `time_unix()`, `zernike()` and `zernikeR()` added.
- Bug fix to the ticking of linked axes.
- Bug fix to the ticking of axes with blank axis tick labels.
- Makefile and configure script improved for portability.

2010 May 19: Pyxplot 0.8.0

Summary:

This release is a major update, for which Pyxplot's original python code has been completely rewritten in C with the addition of many new features. Because of the scale of this update, there is some minor syntax incompatibility with previous versions where features have undergone particularly heavy change. The most apparent change is the increase in speed and efficiency resultant from the use of a compiled language: especially when handling large data files, Pyxplot 0.8.0 can run more than an order-of-magnitude faster than previous versions.

Details:

- The handling of large data files has been streamlined to require around an order-of-magnitude less time and memory.
- Pyxplot's mathematical environment has been extended to handle complex numbers and quantities with physical units.
- The range of mathematical functions built into Pyxplot has been massively extended.
- The `solve` command has been added to allow the solution of systems of equations.
- The `maximize` and `minimize` commands have been added to allow searches for local extrema of functions.
- An `fft` command has been added for performing Fourier transforms on data.
- New plot styles – `filledregion` and `yerrorshaded` – have been added for plotting filled error regions.
- The configuration of linked axes has been entirely redesigned.
- Parametric function plotting has been implemented.

- Colours can now be specified by RGB, HSB or CMYK components, as well as by name.
- Several commands, e.g. `box`, `circle`, `ellipse`, etc., have been added to allow vector graphics to be produced in Pyxplot's multiplot environment.
- The `jpeg` command has been generalised to allow the incorporation of not only `jpeg` images, but also `bmp`, `gif` and `png` images, onto multiplot canvases. The command has been renamed `image` in recognition of its wider applicability. Image transparency is now supported in `gif` and `png` images.
- The `spline` command, now renamed the `interpolate` command, has been extended up provide many types of interpolation between datapoints.
- A wide range of conditional and flow control structures have been added to Pyxplot's command language – these are the `do`, `for`, `foreach`, `if` and `while` commands and the `conditionalS` and `conditionalN` mathematical functions.
- Input filters have been introduced as a mechanism by which datafiles in arbitrary formats can be read.
- Pyxplot's command-line interface now supports tab completion.
- The `show` command has been reworked to produce pastable output.
- Many minor bugs have been fixed.

2009 Nov 17: Pyxplot 0.7.1

Summary:

This release has no major new features, but fixes several serious bugs in version 0.7.0.

Details:

- The `exec` command did not work in Pyxplot 0.7.0; this issue has been resolved.
- The `xyerrorrange` plot style did not work in Pyxplot 0.7.0; this issue has been resolved.
- Pyxplot 0.7.0 produces large numbers of python deprecation error messages when run under python 2.6; the code has been updated to remove references to deprecated python functions.

Details – Change of System Requirements:

- In order to fix some of the bugs listed above, it has been necessary to fix bugs in the PyX graphics library as well as those in Pyxplot. As a result, and to ensure that these bugfixes reach users as quickly as possible, we have opted to ship our own modified version of PyX 0.10, called dcfPyX with Pyxplot.

2008 Oct 14: Pyxplot 0.7.0

Summary:

Third Pyxplot beta-release. The code has undergone significant streamlining, and now runs approximately twice as fast as version 0.6.3 when handling large datafiles. Memory usage has also been radically reduced. Two new data processing commands have been introduced. The `tabulate` command can be used to produce textual datafiles, allowing the user to read data in from files, apply some analysis, and then write the processed data back to file. The `histogram` command can be used to estimate the frequency densities of sets of data points, either by binning them into a bar chart, or by fitting a functional form to their frequency density.

Details – New and Extended Commands:

- `tabulate`
- `histogram`
- `set label` and `text` commands extended to allow a color to be specified.

Details – API changes

- `diff_dx()` and `int_dx()` functions – the function to be differentiated or integrated must now be placed in quotation marks.

Details – Change of System Requirements:

- Requirement of PyX version 0.9 has been updated to PyX version 0.10. Note that new versions of the PyX graphics library are not generally backwardly compatible.

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