The GNU Plotting Utilities

Programs and functions for drawing and data plotting
Version 2.2

Robert S. Maier and Nicholas B. Tufillaro

1 The GNU Plotting Utilities

The GNU plotting utilities consist of eight command-line programs: the graphics programs `graph`, `plot`, `pic2plot`, `tek2plot`, and `plotfont`, and the mathematical programs `spline`, `ode`, and `double`. Distributed with these programs is GNU `libplot`, the library on which the graphics programs are based. `libplot` is a function library for device-independent two-dimensional vector graphics, including vector graphics animations under the X Window System. There are versions for both C and C++.

The graphics programs and `libplot` can export vector graphics in the following ten formats.

X
: If this output option is selected, there is no output file. Output is directed to a popped-up window on an X Window System display.

PNM
: This is “portable anymap” format. There are three types of portable anymap: PBM (portable bitmap, for monochrome images), PGM (portable graymap), and PPM (portable pixmap, for colored images). The output file will be in whichever of these three formats is most appropriate. Portable anymaps may be viewed or edited with many applications, such as the free image display application `xv`.

GIF
: This is pseudo-GIF format rather than true GIF format. Unlike GIF format it does not use LZW compression, so it does not transgress the Unisys LZW patent. However, files in pseudo-GIF format may be viewed or edited with any application that understands GIF format, such as `xv`.

AI
: This is the format used by Adobe Illustrator. Files in this format may be edited with Adobe Illustrator (version 5, and more recent versions), or other applications.

PS
: This is `idraw`-editable Postscript format. Files in this format may be sent to a Postscript printer, imported into another document, or edited with the free `idraw` drawing editor. See Section D.1 [idraw], page 149.

Fig
: This is a graphics format that may be displayed or edited with the free `xfig` drawing editor. See Section D.2 [xfig], page 149.

PCL 5
: This is a powerful version of Hewlett–Packard’s Printer Control Language. Files in this format may be sent to a LaserJet printer or compatible device (note that most inkjets do not support PCL 5).

HP-GL
: This is Hewlett–Packard’s Graphics Language. By default, the modern variant HP-GL/2 is produced. Files in HP-GL or HP-GL/2 format may be imported into a document or sent to a plotter.

Tek
: This is the graphics format understood by Tektronix 4014 terminals and emulators, including the emulators built into the `xterm` terminal emulator program and the MS-DOS version of `kermit`.

Metafile
: This is device-independent GNU graphics metafile format. The `plot` program can translate it to any of the preceding formats.

Of the command-line graphics programs, the best known is graph, which is an application for plotting two-dimensional scientific data. It reads one or more data files containing datasets, and outputs a plot. The above output formats are supported. The corresponding commands are graph -T X, graph -T pnm, graph -T gif, graph -T ai, graph -T ps, graph -T fig, graph -T pcl, graph -T hpgl, graph -T tek, and graph. graph without a '-T' option (referred to as 'raw graph') produces output in GNU metafile format.

graph can read datasets in both ASCII and binary format, and datasets in the 'table' format produced by the plotting program gnuplot. It produces a plot with or without axes and labels. You may specify labels and ranges for the axes, and the size and position of the plot on the display. The labels may contain subscripts and subscripts, Greek letters, and other special symbols; there is also support for Cyrillic script (i.e., Russian) and Japanese. You may specify the type of plotting symbol used for each dataset, and such parameters as the style and thickness of the line (if any) used to connect points in a dataset. The plotting of filled regions is supported, as is the drawing of error bars. graph provides full support for multiplotting. With a single invocation of graph, you may produce a plot consisting of many sub-plots, either side by side or inset. Each sub-plot will have its own axes and data.

graph -T X, graph -T tek, and raw graph have a feature that most plotting programs do not have. They can accept input from a pipe, and plot data points to the output in real time. For this to occur, the user must specify ranges for both axes, so that graph does not need to wait until the end of the input before determining them.

The plot program is a so-called plot filter. It can translate GNU graphics metafiles (produced for example by raw graph) into any supported output format. The corresponding commands are plot -T X, plot -T pnm, plot -T gif, plot -T ai, plot -T ps, plot -T fig, plot -T pcl, plot -T hpgl, plot -T tek, and plot. The plot program is useful if you wish to produce output in several different formats while invoking graph only once. It is also useful if you wish to translate files in the traditional 'plot(5)' format produced by, e.g., the non-GNU versions of graph provided with some operating systems. GNU metafile format is compatible with plot(5) format.

The pic2plot program can translate from the pic language to any supported output format. The pic language, which was invented at Bell Laboratories, is used for creating box-and-arrow diagrams of the kind frequently found in technical papers and textbooks. The corresponding commands are pic2plot -T X, pic2plot -T pnm, pic2plot -T gif, pic2plot -T ai, pic2plot -T ps, pic2plot -T fig, pic2plot -T pcl, pic2plot -T hpgl, pic2plot -T tek, and pic2plot.

The tek2plot program can translate from Tektronix format to any supported output format. The corresponding commands are tek2plot -T X, tek2plot -T pnm, tek2plot -T gif, tek2plot -T ai, tek2plot -T ps, tek2plot -T fig, tek2plot -T pcl, tek2plot -T hpgl, and tek2plot. tek2plot is useful if you have an older application that produces drawings in Tektronix format.

The plotfont program is a simple utility that displays a character map for any font that is available to graph, plot, pic2plot, or tek2plot. The 35 standard Postscript fonts are available if the '-T X', '-T ai', '-T ps', or '-T fig' options are used. The 45 standard PCL 5 fonts (i.e., "LaserJet" fonts) are available if the '-T ai', '-T pcl' or '-T hpgl' options are used. In the latter two cases ('-T pcl' and '-T hpgl'), a number of Hewlett–Packard vector fonts are available as well. A set of 22 Hershey vector fonts, including Cyrillic fonts and a Japanese font, is always available. When producing output for an X Window System display, any of the graphics programs can use scalable X fonts.

Of the command-line mathematical programs, `spline` does spline interpolation of scalar or vector-valued data. It normally uses either cubic spline interpolation or exponential splines in tension, but like `graph` it can function as a real-time filter under some circumstances. Besides splining datasets, it can construct curves, either open or closed, through arbitrarily chosen points in d-dimensional space. `ode` provides the ability to integrate an ordinary differential equation or a system of ordinary differential equations, when provided with an explicit expression for each equation. It supplements the plotting program `gnuplot`, which can plot functions but not integrate ordinary differential equations. The final command-line mathematical program, `double`, is a filter for converting, scaling and cutting binary or ASCII data streams. It is still under development and is not yet documented.

The `libplot` function library is discussed at length elsewhere in this documentation. It gives C and C++ programs the ability to draw such objects as lines, open and closed polylines, arcs (both circular and elliptic), quadratic and cubic Bezier curves, circles and ellipses, points, marker symbols, and text strings. The filling of objects other than points, marker symbols, and text strings is supported (fill color, as well as pen color, can be set arbitrarily). Text strings can be drawn in any of a large number of fonts. The 35 standard Postscript fonts are supported by the X Window System, Illustrator, Postscript, and `xfig` drivers, and the 45 standard PCL 5 fonts are supported by the Illustrator, PCL 5 and HP-GL/2 drivers. The latter two also support a number of Hewlett–Packard vector fonts. All drivers, including the PNM, GIF, Tektronix and metafile drivers, support a set of 22 Hershey vector fonts.

The support for drawing text strings is extensive. Text strings may include subscripts and superscripts, and may include characters chosen from more than one font in a typeface. Many non-alphanumeric characters may be included. The entire collection of over 1700 'Hershey glyphs' digitized by Allen V. Hershey at the U.S. Naval Surface Weapons Center, which includes many curious symbols, is built into `libplot`. Text strings in the so-called EUC-JP encoding (i.e., the Extended Unix Code for Japanese) can be also be drawn. Such strings may include both syllabic Japanese characters (Hiragana and Katakana) and ideographic Japanese characters (Kanji). A library of 603 Kanji, including 596 of the 2965 frequently used Level 1 Kanji, is built into `libplot`.

The drawing editors `idraw` and `xfig` are not distributed along with the GNU plotting utilities. However, they are free software, and you may readily obtain them elsewhere (see Appendix D [Auxiliary Software], page 149).

2 The graph Application

Each invocation of **graph** reads one or more datasets from files named on the command line or from standard input, and prepares a plot. There are many command-line options for adjusting the visual appearance of the plot. See Section 2.6 [graph Invocation], page 13, for documentation on all options. The following sections explain how to use the most frequently used options, by giving examples.

2.1 Simple examples using graph

By default, **graph** reads ASCII data from the files specified on the command line, or from standard input if no files are specified. The data are pairs of numbers, interpreted as the x and y coordinates of data points:

```
0.0  0.0
1.0  0.2
2.0  0.0
3.0  0.4
4.0  0.2
5.0  0.6
```

Data points do not need to be on different lines, nor do the x and y coordinates of a data point need to be on the same line. However, there should be no blank lines in the input if it is to be viewed as forming a single dataset.

To plot such a dataset with **graph**, you could do

```
graph -T ps ascii_data_file > plot.ps
```

or equivalently

```
graph -T ps < ascii_data_file > plot.ps
```

This will produce an encapsulated Postscript file `plot.ps`, which you may include in another document, display on a screen, or send directly to a printer. (The '`--page-size`' option, or the `PAGESIZE` environment variable, specifies the size of the printed page. The default is "letter", i.e., 8.5 in by 11 in, but "a4" or other ISO or ANSI page sizes can be specified instead.)

You may also do

```
graph -T fig < ascii_data_file > plot.fig
```

to produce a file `plot.fig` that you may edit with the the free `xfig` drawing editor, or

```
graph -T ai < ascii_data_file > plot.ai
```

to produce a file `plot.ai` that you may edit with Adobe Illustrator. If you do

```
graph -T hpgl < ascii_data_file > plot.plt
```

you will produce a file `plot.plt` in the Hewlett–Packard Graphics Language (HP-GL/2) that you may send to a Hewlett–Packard plotter. Similarly, you may use `graph -T pcl` to produce a file in PCL 5 format that may be printed on a LaserJet or other laser printer.

You may use `graph -T X` to pop up a window on an X Window System display, and display the plot in it. For that, you would do

```
graph -T X < ascii_data_file
```

If you use `graph -T X`, no output file will be produced; only a window. The window will vanish if you type '`q`' or click your mouse in it.

You may also use `graph -T pnm` to produce a PNM file (a "portable anymap"), and `graph -T gif` to produce a pseudo-GIF file. If the free image display application `xv` is available on your system, you would use either of the two commands

```
graph -T pnm < ascii_data_file | xv -
graph -T gif < ascii_data_file | xv -
```

to display the output file.

Another thing you can do is use `graph -T tek` to display a plot on a device that can emulate a Tektronix 4014 graphics terminal. `xterm`, the X Window System terminal emulator, can do this. Within an `xterm` window, you would do

```
graph -T tek < ascii_data_file
```

`xterm` normally emulates a VT100 terminal, but when this command is issued from within it, it will pop up a second window (a 'Tektronix window') and draw the plot in it. The Japanese terminal emulator `kterm` should be able to do the same, provided that it is correctly installed. Another piece of software that can emulate a Tektronix 4014 terminal is the MS-DOS version of `kermit`.

`graph` may behave differently depending on the environment in which it is invoked. We have already mentioned the `PAGESIZE` environment variable, which affects the operation of `graph -T ai`, `graph -T ps`, `graph -T fig`, `graph -T pcl`, and `graph -T hpgl`. Similarly, the `BITMAPSIZE` environment variable affects the operation of `graph -T X`, `graph -T pnm`, and `graph -T gif`. The `DISPLAY` environment variable affects the operation of `graph -T X`, and the `TERM` environment variable affects the operation of `graph -T tek`. There are also several environment variables that affect the operation of `graph -T pcl` and `graph -T hpgl`. For a complete discussion of the effects of the environment on `graph`, see Section 2.7 [graph Environment], page 24. The following remarks apply irrespective of which output format is specified.

By default, successive points in the dataset are joined by solid line segments, which form a polygonal line or polyline that we call simply a 'line'. You may choose the style of line (the 'linemode') with the '`-m`' option:

```
graph -T ps -m 2 < ascii_data_file > plot.ps
```

Here '`-m 2`' indicates that linemode #2 should be used. If the dataset is rendered in monochrome, which is the default, the line can be drawn in one of five distinct styles. Linemodes #1 through #5 signify solid, dotted, dotdashed, shortdashed, and longdashed; thereafter the sequence repeats. If the '`-C`' option is used, the dataset will be rendered in color. For colored datasets, the line can be drawn in one of 25 distinct styles. Linemodes #1 through #5 signify red, green, blue, magenta, and cyan; all are solid. Linemodes #6 through #10 signify the same five colors, but dotted rather than solid. Linemodes #11 through #16 signify the same five colors, but dotdashed, and so forth. After linemode #25, the sequence repeats. Linemode #0, irrespective of whether the rendering is in monochrome or color, means that the line is not drawn.

If you wish to *fill* the polygon bounded by the line (i.e., shade it, or fill it with a solid color), you may use the '`-q`' option. For example,

```
echo .1 .1 .1 .9 .9 .9 .9 .1 .1 .1 | graph -T ps -C -m 1 -q 0.3 > plot.ps
```

will plot a square region with vertices (0.1,0.1), (0.1,0.9), (0.9,0.9), and (0.9,0.1). The repetition of the first vertex (0.1,0.1) at the end of the sequence of vertices ensures that the square will be closed: all four segments of its boundary will be drawn. The square will be drawn in red, since the colored version of linemode #1 is requested. The interior of the square will be filled with red to an intensity of 30%, as the '-q 0.3' option specifies. If the intensity were zero, the region would be filled with white, and if it were 1.0, the region would be filled with solid color. If the intensity were negative, the region would be unfilled, or transparent (the default).

You may choose the thickness ('width') of the line, whether it is filled or not, by using the '-W' option. For example, '-W 0.01' specifies that the line should have a thickness equal to 0.01 times the size of the display. Also, you may put symbols at each data point along the line by doing, for example,

```
graph -T ps -S 3 0.1 < ascii_data_file > plot.ps
```

where the first argument 3 indicates which symbol to plot. The optional second argument 0.1 specifies the symbol size as a fraction of the size of the 'plotting box': the square within which the plot is drawn. Symbol #1 is a dot, symbol #2 is a plus sign, symbol #3 is an asterisk, symbol #4 is a circle, symbol #5 is a cross, and so forth. (See Section A.5 [Marker Symbols], page 144.) Symbols 1 through 31 are the same for all display types, and the color of a symbol will be the same as the color of the line it is plotted along.

Actually, you would probably not want to plot symbols at each point in the dataset unless you turn off the line joining the points. For this purpose, the 'negative linemode' concept is useful. A line whose linemode is negative is not visible; however, any symbols plotted along it will have the color associated with the corresponding positive linemode. So, for example,

```
graph -T ps -C -m -3 -S 4 < ascii_data_file > plot.ps
```

will plot a blue circle at each data point. The circles will not be joined by line segments. By adding the optional second argument to the '-S' option, you may adjust the size of the circles.

graph will automatically generate abscissa (i.e., x) values for you if you use the '-a' option. If this option is used, no abscissa values should be given in the data file. The data points will be taken to be regularly spaced along the abscissa. The two arguments following '-a' on the command line will be taken as the sampling interval and the abscissa value of the first data point. If they are absent, they default to 1.0 and 0.0 respectively. For example, the command

```
echo 0 1 0 | graph -T ps -a > plot.ps
```

produces exactly the same plot as

```
echo 0 0 1 1 2 0 | graph -T ps > plot.ps
```

graph will plot data with error bars, if the '-I e' option is specified. If it is, the dataset should consist of triples (x,y,*error*) rather than pairs (x, y). A vertical error bar of the appropriate length will be plotted at each data point. You may plot a symbol at each data point, along with the error bar, by using the '-S' option in the usual way. The symbol will be the same for each point in the dataset. You may use the '-a' option in conjunction with '-I e', if you wish. If you do, the dataset should contain no abscissa (i.e., x) values.

By default the limits on the x and y axes, and the spacing between the labeled ticks on each axis, are computed automatically. You may wish to set them manually. You may accomplish this with the '-x' and '-y' options.

```
echo 0 0 1 1 2 0 | graph -T ps -x -1 3 -y -1 2 > plot.ps
```

will produce a plot in which the x axis extends from -1 to 3, and the y axis from -1 to 2. By default, graph tries to place about six numbered ticks on each axis. By including an optional third argument to either '-x' or '-y', you may manually set the spacing of these ticks, also. For example, using '-y -1 2 1' rather than '-y -1 2' will produce a y axis with labeled ticks at -1, 0, 1, and 2, rather than at the locations that graph would choose by default, which would be -1, -0.5, 0, 0.5, 1, 1.5, and 2. In general, if a third argument is present then labeled ticks will be placed at each of its integer multiples.

To make an axis logarithmic, you may use the '-l' option. For example,

```
echo 1 1 2 3 3 1 | graph -T ps -l x > plot.ps
```

will produce a plot in which the x axis is logarithmic, but the y axis is linear. To make both axes logarithmic, you would use '-l x -l y'. By default, the upper and lower limits on a logarithmic axis are powers of ten, and there are tick marks at each power of ten and at its integer multiples. The tick marks at the powers of ten are labeled. If the axis spans more than five orders of magnitude, the tick marks at the integer multiples are omitted.

If you have an unusually short logarithmic axis, you may need to increase the number of labeled ticks. To do this, you should specify a tick spacing manually. For example, '-l x -x 1 9 2' would produce a plot in which the x axis is logarithmic and extends from 1 to 9. Labeled ticks would be located at each integer multiple of 2, i.e., at 2, 4, 6, and 8.

You may label the x and y axes with the '-X' and '-Y' options, respectively. For example,

```
echo 1 1 2 3 3 1 | graph -T ps -l x -X "A Logarithmic Axis" > plot.ps
```

will label the log axis in the preceding example. By default the label for the y axis (if any) will be rotated 90 degrees, unless you use the '--toggle-rotate-y-label' option. You may specify a 'top label', or title for the plot, by using the '-L' option. Doing, for example,

```
echo 1 1 2 3 3 1 | graph -T ps -l x -L "A Simple Example" > plot.ps
```

will produce a plot with a title on top.

The size of the x axis and y axis labels is specified with the '-f' option, and the size of the title is specified with the '--title-font-size' option. For example,

```
echo 1 1 2 3 3 1 | graph -T ps -X "Abscissa" -f 0.1 > plot.ps
```

will produce a plot in which the font size of the x axis label, and each of the numerical tick labels, is very large (0.1 times the size of the plotting box, i.e., the square within which the plot is drawn).

The font in which the labels specified with the '-X', '-Y', and '-L' options are drawn can be specified with the '-F' option. For example, '-F Times-Roman' will make the labels appear in Times-Roman instead of the default font (which is Helvetica, unless '-T pnm', '-T gif', '-T pcl', '-T hpgl' or '-T tek' is specified). Font names are case-insensitive, so '-F times-roman' will work equally well. The available fonts include 35 Postscript fonts (for all variants of graph other than graph -T pnm, graph -T gif, graph -T pcl, graph -T hpgl and graph -T tek), 45 PCL 5 fonts (for graph -T ai, graph -T pcl and graph -T hpgl), a number of Hewlett–Packard vector fonts (for graph -T pcl and graph -T hpgl), and 22 Hershey vector fonts. The Hershey fonts include HersheyCyrillic, for Russian, and HersheyEUC, for Japanese. For a discussion of the available fonts, see Section A.1 [Text Fonts], page 124. The `plotfont` utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

The format of the labels drawn with the '`-X`', '`-Y`', and '`-L`' options may be quite intricate. Subscripts, superscripts, square roots, and switching fonts within a typeface are all allowed. The above examples do not illustrate this, but for details, see Section A.4 [Text String Format], page 131.

Each of the preceding examples produced a plot containing the default sort of grid (a square box, with ticks and labels drawn along its lower edge and its left edge). There are actually several sorts of grid you may request. The '`-g 0`', '`-g 1`', '`-g 2`', and '`-g 3`' options yield successively fancier grids. What they yield, respectively, is no grid at all, a pair of axes with ticks and labels, a square box with ticks and labels, and a square box with ticks, labels, and grid lines. As you can see, '`-g 2`' is the default. There is also a '`-g 4`' option, which yields a slightly different sort of grid: a pair of axes that cross at the origin. This last sort of grid is useful if the x or y coordinates of the data points you are plotting are both positive and negative.

2.2 Rotated, non-square and displaced plots

To rotate the plotting box by 90 degrees counterclockwise on your graphics display, you would add '`--rotation 90`' to the `graph` command line. You may also specify '`--rotation 180`', to produce an upside-down plot, or '`--rotation 270`'.

To alter the linear dimensions of the plotting box, and also to position it in a different part of the display, you could do something like

```
graph -T ps -h .3 -w .6 -r .1 -u .1 < ascii_data_file > plot.ps
```

Here the '`-h`' and '`-w`' options specify the height and width of the plotting box, and the '`-r`' and '`-u`' options indicate how far up and to the right the lower left corner of the plotting box should be positioned. All dimensions are expressed as fractions of the size of the graphics display, which by convention is a square. By default, the height and width of the plotting box equal 0.6, and the 'upward shift' and the 'rightward shift' equal 0.2. So the above example will produce a plot that is half as tall as usual. Compared to its usual position, the plot will be shifted slightly downward and to the left.

Several command-line options specify sizes or dimensions as fractions of the size of the plotting box, rather than as fractions of the size of the display. For example, '`-S 3 .01`' specifies that the plotting symbols for the following dataset should be of type #3, and should have a font size equal to 0.01, i.e. 0.01 times the minimum dimension (height or width) of the plotting box. If the '`-h`' or '`-w`' options are employed to expand or contract the plot, such sizes or dimensions will scale in tandem. That is presumably the right thing to do.

The '`-h`', '`-w`', '`-r`', and `-u` options may be combined with the '`--rotation`' option. If they appear together, the plotting box is first positioned, and then rotated. In fact, '`--rotation`' specifies how the plot should be mapped to the graphics display, rather than how the plot is designed.

The 'graphics display' is an abstraction. For `graph -T X`, it is a window on an X display. For `graph -T pnm` and `graph -T gif`, it is a square or rectangular bitmap. In these three cases, the size of the graphics display can be set by using the `--bitmap-size` option, or by setting the `BITMAPSIZE` environment variable. For `graph -T tek`, the graphics display is a square region occupying the central part of a Tektronix display. (Tektronix displays are 4/3 times as wide as they are high.) For `graph -T ai` and `graph -T ps`, by default it is a square region centered on an 8.5 in by 11 in page (US letter size), occupying its full width with allowance being made for margins. For `graph -T fig`, by default it is a square region of the same size, positioned in the upper left corner of an `xfig` display. For `graph -T pcl` and `graph -T hpgl`, by default it is a square region of the same size, with position and orientation on the page being controlled by environment variables. The page size used by `graph -T ai`, `graph -T ps`, `graph -T fig`, `graph -T pcl`, and `graph -T hpgl` can be set by using the `--page-size` option, or by setting the environment variable `PAGESIZE`. For example, setting `PAGESIZE` to `"a4"` would position the graphics display appropriately on an A4-size page (21 cm by 29.7 cm).

2.3 Preparing a plot from more than one dataset

It is frequently the case that several datasets need to be displayed on the same plot. If so, you may wish to distinguish the points in different datasets by joining them by lines of different types, or by using plotting symbols of different types.

A more complicated example would be the following. You may have a file containing a dataset that is the result of experimental observations, and a file containing closely spaced points that trace out a theoretical curve. The second file is a dataset in its own right. You would presumably plot it with line segments joining successive points, so as to trace out the theoretical curve. But the first dataset, resulting from experiment, would be plotted without such line segments. In fact, a plotting symbol would be plotted at each of its points.

These examples, and others like them, led us to define a set of seven *attributes* which define the way in which a dataset should be plotted. These attributes, which can be set by command-line options, are the following.

1. color/monochrome
2. linemode
3. linewidth
4. symbol type
5. symbol size
6. symbol font name
7. fill fraction

Color/monochrome (a choice of one or the other) is the simplest. This choice is toggled with the '`-C`' option. The 'linemode' (i.e., line style) specifies how the line segments joining successive points should be drawn; it is specified with the '`-m`' option. Linemode #0 means no linemode at all, for example. 'Linewidth' means line thickness; it is specified with the '`-W`' option. 'Symbol type' and 'symbol size', which are specified with the '`-S`' option, specify the symbol plotted at each point of the dataset. 'Symbol font name' refers to the font from which plotting symbols #32 and above, which are taken to be characters rather than geometric symbols, are selected. It is set with the '--symbol-font-name' option, and is relevant only if '-S' is used to request such special plotting symbols. Finally, the polygonal line joining the points in a dataset may be *filled*, to create a filled or shaded polygon. The 'fill fraction' is set with the '-q' option. A negative fill fraction means no fill, or transparent; zero means white, and 1.0 means solid, or fully colored.

The preceding seven attributes refer to the way in which datasets are plotted. Datasets may also differ from one another in the way in which they are read from files. The dataset(s) in a file may or may not contain error bars, for example. If a file contains data with error bars, the '-I e' option should occur on the command line before the file name. (The '-I' option specifies the input format for the following files.)

The following illustrates how datasets in three different input files could be plotted simultaneously.

```
graph -T ps -m 0 -S 3 file_1 -C -m 3 file_2 -C -W 0.02 file_3 > output.ps
```

The dataset in file_1 will be plotted in linemode #0, so successive points will not be joined by lines. But symbol #3 (an asterisk) will be plotted at each point. The dataset in file_2 will be plotted in color, and linemode #3 will be used. In color plotting, linemode #3 is interpreted as a solid blue line. The second '-C' on the command line turns off color for file_3. The points in the third dataset will be joined by a black line with thickness 0.02, as a fraction of the size (i.e., minimum dimension) of the graphics display.

The above command line could be made even more complicated by specifying additional options (e.g., '-q' or '-I') before each file. In fact the command line could also include such standard options as '-x' or '-y', which specify the range of each axis. Such options, which refer to the plot as a whole rather than to individual datasets, should appear before the first file name. For example, you could do

```
graph -T ps -x 0 1 0.5 -m 0 -S 3 file_1 -C -m 3 file_2 > output.ps
```

Note that it is possible to include the special file name '-', which refers to standard input, on the command line. So you may produce a plot in part from files, and in part from input that is piped to graph from another program.

Each input file may include more than one dataset. If so, the command line options preceding a file on the command line will take effect for all datasets in that file. There are two exceptions to this. By default, the linemode is incremented ('bumped') from one dataset to the next. This feature is usually quite convenient. For example, if you do

```
graph -T ps -m 3 file_1 > output.ps
```

the first dataset in file_1 will appear in linemode #3, the second in linemode #4, etc. In fact if you do

```
graph -T ps file_1 file_2 ... > output.ps
```

without specifying linemode explicitly, the successive datasets read from the files on the command line will appear in linemode #1, linemode #2, ... If you do not like this feature, you may turn it off, or in general toggle it, by using the '-B' option.

You may also control manually the linemode and symbol type used for the datasets within any file. You would do this by including directives in the file itself, rather than on the command line. For example, if the line

```
#m=-5,S=10
```

appeared in an ASCII-format input file, it would be interpreted as a directive to switch to linemode #−5 and symbol type #10 for the following dataset. Future releases of graph may provide the ability to set each of the seven dataset attributes in this way.

2.4 Multiplotting: placing multiple plots on a single page

It is occasionally useful to display several plots at once on a single page, or on a single graphics display. We call such a composite plot a *multiplot*. One common sort of multiplot is a small plot inset into a larger one. Another sort is two or more plots side by side.

graph can draw multiplots consisting of an arbitrarily large number of sub-plots. When multiplotting, graph draws each sub-plot in its own 'virtual display'. When an ordinary plot is drawn, the virtual display is the same as the physical display. But when a multiplot is drawn, the virtual display may be any smaller square region. The following example illustrates the idea.

```
graph -T ps data_file_1 --reposition .35 .35 .3 data_file_2
```

Here data_file_1 is plotted in the usual way. The '--reposition' option specifies that when data_file_2 is plotted, it will be drawn within a virtual display. For the purposes of the '--reposition' option, the physical display is a square with lower left corner (0.0,0.0) and upper right corner (1.0,1.0). In those coordinates, the virtual display will be a square of size 0.3 with lower left corner (0.35,0.35). So the second sub-plot will be inset into the first.

Just as the '-w', '-h', '-r', and '-u' options may be used to set the size and position of a plotting box within the physical display, so they may be used to set the size and position of a plotting box within a virtual display. For example,

```
graph -T ps data_file_1 --reposition .35 .35 .3 -w .4 -r .3 data_file_2
```

will yield a multiplot in which the second sub-plot is significantly different. Its plotting box will have a width only 0.4 times the width of the virtual display. However, the plotting box will be centered within the virtual display, since the distance between the left edge of the plotting box and the left edge of the virtual display will be 0.3 times the width of the virtual display.

By convention, before each sub-plot of a multiplot other than the first is drawn, a 'blankout region' surrounding its plotting box is erased. (That is, it is filled with white.) This erasure prevents the sub-plots from overlapping and producing a messy result. By default, the blankout region is a rectangular region 30% larger in each dimension than the plotting box for the sub-plot. This is appropriate if the sub-plot is a small one that is inset into the first sub-plot. It may not be appropriate, however, if you are preparing a multiplot in which several sub-plots appear side by side. You may use the '--blankout' option to adjust this parameter. For example, specifying '--blankout 1.0' will make the blankout region for a sub-plot coincide with its plotting box. Specifying '--blankout 0.0' will prevent any blanking out from occurring. The blankout parameter may differ from sub-plot to sub-plot.

It should be emphasized that every sub-plot in a multiplot is a plot in its own right. All the usual options ('-m', '-S', '-x', '-y', etc.) can be applied to each sub-plot separately. The options for a sub-plot should occur on the graph command line immediately after the '--reposition' option that applies to it. Each sub-plot may be prepared from more than a single dataset, also. The names of the data files for each subplot should occur on the command line before the following '--reposition' option, if any.

2.5 Reading binary and other data formats

By default, graph reads datasets in ASCII format. But it can also read datasets in any of three binary formats (single precision floating point, double precision floating point, and integer). These three input formats are specified by the '-I d', '-I f', and '-I i' options, respectively.

There are two advantages to using binary data: 1) graph runs significantly faster because the computational overhead for converting data from ASCII to binary is eliminated, and 2) the input files may be significantly smaller. If you have very large datasets, using binary format may reduce storage and runtime costs.

For example, you may create a single precision binary dataset as output from a C language program:

```
#include <stdio.h>
void write_point (float x, float y)
{
  fwrite(&x, sizeof (float), 1, stdout);
  fwrite(&y, sizeof (float), 1, stdout);
}
```

You may plot data written this way by doing:

```
graph -T ps -I f < binary_data_file > plot.ps
```

The inclusion of multiple datasets within a single binary file is supported. If a binary file contains more than a single dataset, successive datasets should be separated by a single occurrence of the the largest possible number. For single precision datasets this is the quantity FLT_MAX, for double precision datasets it is the quantity DBL_MAX, and for integer datasets it is the quantity INT_MAX. On most machines FLT_MAX is approximately 3.4×10^{38}, DBL_MAX is approximately 1.8×10^{308}, and INT_MAX is $2^{31} - 1$.

If you are reading datasets from more than one file, it is not required that the files be in the same format. For example,

```
graph -T ps -I f binary_data_file -I a ascii_data_file > plot.ps
```

will read binary_data_file in 'f' (binary single precision) format, and ascii_data_file in 'a' (normal ASCII) format.

There is currently no support for reading and plotting binary data with error bars. If you have data with error bars, you should supply the data to graph in ASCII, and use the '-I e' option.

graph can also read data files in the ASCII 'table' format produced by the gnuplot plotting program. For this, you should use the '-I g' option. Such a data file may consist of more than one dataset.

To sum up: there are six supported data formats, 'a' (normal ASCII), 'e' (ASCII with error bars), 'g' (the ASCII 'table' format produced by gnuplot), 'f' (binary single precision), 'd' (binary double precision), and 'i' (binary integer). Input files may be in any of these six formats.

2.6 `graph` command-line options

The `graph` program reads one or more datasets from files named on the command line or from standard input, and prepares a plot. The output format or display type is specified with the '`-T`' option.

By default, `graph` reads ASCII data from the files specified on the command line. The data are pairs of numbers, interpreted as the x and y coordinates of data points. If no file names are specified, or the file name '`-`' is specified, the standard input is read. The output file is written to standard output, unless the '`-T X`' option is specified. In that case the graph is displayed in a window on an X Window System display, and there is no output file.

There are many command-line options for adjusting the visual appearance of the plot. The relative order of file names and command-line options is important. Only the options that precede a file name on the command line take effect for that file.

The following sections list the possible options. Each option that takes an argument is followed, in parentheses, by the type and default value of the argument. There are five sorts of option.

1. Options affecting an entire plot. (See Section 2.6.1 [Plot Options], page 13.)
2. Options affecting the reading and drawing of individual datasets within a plot. (See Section 2.6.2 [Dataset Options], page 20.)
3. Options for multiplotting (drawing several sub-plots within a plot). (See Section 2.6.3 [Multiplot Options], page 23.)
4. Options relevant only to raw `graph`, i.e., relevant only if no display type or output format is specified with the '`-T`' option. (See Section 2.6.4 [Raw graph Options], page 23.)
5. Options requesting information (e.g., '`--help`'). (See Section 2.6.5 [Info Options], page 24.)

2.6.1 Plot options

The following options affect an entire plot. They should normally occur at most once, and should appear on the command line before the first file name. If a multiplot is being drawn, they may (with the exception of the '`-T`' option) occur more than once. If so, the second and later occurrences should be placed on the command line immediately after each '`--reposition` x y' option.

'`-T` *type*'
'`--display-type` *type*'

> (String, default "meta".) Select a display type or output format of type *type*, which may be one of the strings "X", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", and "meta". These refer respectively to the X Window System, portable anymap (PBM/PGM/PPM) format, pseudo-GIF format, the format used by Adobe Illustrator, `idraw`-editable Postscript, the format used by the `xfig` drawing editor, the Hewlett–Packard PCL 5 printer language, the Hewlett–Packard Graphics Language (by default, HP-GL/2), Tektronix format, and device-independent GNU graphics metafile format.

'-E $x|y$'
'--toggle-axis-end $x|y$'

Set the position of the indicated axis to be on the other end of the plotting box from what is currently the case. E.g., '-E y' will cause the y axis to appear on the right of the plot rather than the left, which is the default. Similarly, '-E x' will cause the x axis to appear at the top of the plot rather than the bottom. Note that if the x axis appears at the top, no plot title will be drawn (since there will be no room).

'-f *size*'
'--font-size *size*'

(Float, default 0.0525.) Set the size of the font used for the axis and tick labels to be *size*. The size is specified as a fraction of the minimum dimension (width or height) of the plotting box.

'-F *font_name*'
'--font-name *font_name*'

(String, default "Helvetica" except for graph -T pnm, graph -T gif, graph -T pcl, graph -T hpgl, graph -T tek, and raw graph, for all of which "Hershey-Serif" is the default.) Set the font used for the axis and tick labels, and for the plot title (if any), to be *font_name*. The choice of font for the plot title may be overridden with the '--title-font-name' option (see below). Font names are case-insensitive. If the specified font is not available, the default font will be used. Which fonts are available depends on which '-T' option is used. For a list of all fonts, see Section A.1 [Text Fonts], page 124. The plotfont utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

'-g *grid_style*'
'--grid-style *grid_style*'

(Integer in the range 0...4, default 2.) Set the grid style for the plot to be *grid_style*. Grid styles 0 through 3 are progressively more fancy, but style 4 is a somewhat different style.

0. no axes, tick marks or labels.
1. a pair of axes, with tick marks and labels.
2. box around plot, with tick marks and labels.
3. box around plot, with tick marks and labels; also grid lines.
4. axes intersect at the origin, with tick marks and labels.

'-h *height*'
'--height-of-plot *height*'

(Float, default 0.6.) Set the fractional height of the plot with respect to the height of the display (or virtual display, in the case of a multiplot) to be *height*. A value of 1.0 will produce a plotting box that fills the available area. Note that tick marks and labels are outside the plotting box, so that values less than 1.0 are generally used.

'`-k` *length*'
'`--tick-size` *length*'

(Float, default .02.) Set the length of the tick marks on each axis to be *length*. A value of 1.0 produces tick marks whose length is equal to the minimum dimension (width or height) of the plotting box. A negative *length* yields tick marks that extend outside the box, rather than inside.

'`-K` *clip_mode*'
'`--clip-mode` *clip_mode*'

(Integer, default 1.) Set the clip mode for the plot to *clip_mode*. The clip mode is relevant only if data points are being joined by a line, and the line is not being filled to create a filled region (since filled regions are clipped in a fixed way). There are three clip modes: 0, 1, and 2. Clip mode 0 means that a line segment joining two data points will be plotted only if neither point is outside the plotting box. Clip mode 1 means that it will be plotted if no more than one of the two points is outside, and clip mode 2 means that it will be plotted even if both are outside. In all cases the line segment will be clipped to the plotting box.

'`-l` *x*|*y*'
'`--toggle-log-axis` *x*|*y*'

Set the specified axis to be a log axis rather than a linear axis, or vice versa. By default, both axes are linear axes.

'`-L` *top_label*'
'`--top-label` *top_label*'

(String, default empty.) Place the text string *top_label* above the plot, as its 'top label', i.e., title. The string may include escape sequences (see Section A.4 [Text String Format], page 131). The '`--title-font-size`' option may be used to specify the size of the font. The font is normally the same as the font used for labeling axes and ticks, as selected by the '`-F`' option. But this can be overridden with the '`--title-font-name`' option.

'`-N` *x*|*y*'
'`--toggle-no-ticks` *x*|*y*'

Toggle the presence of ticks and tick labels on the specified axis. This applies to the grid styles that normally include ticks and tick labels, i.e., grid styles 1, 2, 3, and 4.

'`-r` *right*'
'`--right-shift` *right*'

(Float, default 0.2.) Move the plot to the right by a fractional amount *right* with respect to the width of the display (or virtual display, in the case of a multiplot). This produces a margin on the left side of the plotting box. A value of 0.5 will produce a margin half the width of the available area. Note that the tick marks and labels are drawn in the margin.

'`-R` *x*|*y*'
'`--toggle-round-to-next-tick` *x*|*y*'

Toggle whether or not the upper and lower limits of the specified axis should be expanded, so that they both become integer multiples of the spacing between labeled tick marks.

This option is meaningful whenever the user specifies either or both of the limits, by using the '`-x`' or '`-y`' option. If the user leaves both limits unspecified, they will always be chosen to satisfy the 'integer multiple' constraint.

'`-s`'
'`--save-screen`'

Save the screen. This option requests that `graph` not erase the display device before it begins to plot.

This option is relevant only to `graph -T tek` and raw `graph`. Tektronix displays and emulators are persistent, in the sense that previously drawn graphics remain visible. So by repeatedly using `graph -T tek -s`, you can build up a multiplot.

'`-t`'
'`--toggle-transpose-axes`'

Transpose the abscissa and ordinate. This causes the axes to be interchanged, and the options that apply to each axis to be applied to the opposite axis. That is, data points are read in as (y, x) pairs, and such options as '`-x`' and '`-X`' apply to the y axis rather than the x axis. If the '`-I e`' option is in force, so that the data points are read with error bars, the orientation of the error bars will be switched between vertical and horizontal.

'`-u` *up*'
'`--upward-shift` *up*'

(Float, default 0.2.) Move the plot up by a fractional amount *up* with respect to the height of the display (or virtual display, in the case of a multiplot). This produces a margin below the plotting box. A value of 0.5 will produce a margin half the height of the available area. Note that the tick marks and labels are drawn in the margin.

'`-w` *width*'
'`--width-of-plot` *width*'

(Float, default 0.6.) Set the fractional width of the plot with respect to the width of the display (or virtual display, in the case of a multiplot) to be *width*. A value of 1.0 will produce a plotting box that fills the available area. Note that the tick marks and labels are outside the plotting box, so values less than 1.0 are generally used.

'`-x [`*lower_limit* `[`*upper_limit* `[`*spacing*`]]]`'
'`--x-limits [`*lower_limit* `[`*upper_limit* `[`*spacing*`]]]`'

(Floats.) The arguments *lower_limit* and *upper_limit* specify the limits of the x axis, and the optional argument *spacing* specifies the spacing of labeled ticks along the axis. If any of the three arguments is missing or is supplied as '`-`' (i.e., as a single hyphen), it is computed from the data. Both arguments *lower_limit* and *upper_limit* must be present if `graph` is to act as a real-time filter.

By default, the supplied limit(s) are strictly respected. However, the '-R x' option may be used to request that they be rounded to the nearest integer multiple of the spacing between labeled ticks. The lower limit will be rounded downward, and the upper limit upward.

'-X *x_label*'
'--x-label *x_label*'
: (String, default empty.) Set the label for the x axis to be the text string *x_label*. The string may include escape sequences (see Section A.4 [Text String Format], page 131). The '-F' and '-f' options may be used to specify the name of the font and the size of the font.

'-y [*lower_limit* [*upper_limit* [*spacing*]]]'
'--y-limits [*lower_limit* [*upper_limit* [*spacing*]]]'
: (Floats.) The arguments specify the limits of the y axis, and the spacing of labeled ticks along it, as for the x axis (see above). Both arguments *lower_limit* and *upper_limit* must be present if `graph` is to act as a real-time filter.

By default, the supplied limit(s) are strictly respected. However, the '-R y' option may be used to request that they be rounded to the nearest multiple of the tick spacing. The lower limit will be rounded downward, and the upper limit upward.

'-Y *y_label*'
'--y-label *y_label*'
: (String, default empty.) Set the label for the y axis to be the text string *y_label*. The string may include escape sequences (see Section A.4 [Text String Format], page 131). The label will be rotated by 90 degrees so that it is parallel to the axis, unless the '--toggle-rotate-y-label' option is used. Some old X Window System displays do not support rotated labels, and require the '--toggle-rotate-y-label' option. The '-F' and '-f' options can be used to specify the name of the font and the size of the font.

'--bg-color *name*'
: (String, default "white".) Set the color used for the plot background to be *name*. This is relevant only to `graph -T X`, `graph -T pnm`, and `graph -T gif`. An unrecognized name sets the color to the default. For information on what names are recognized, see Appendix B [Color Names], page 146. The environment variable `BG_COLOR` can equally well be used to specify the background color.

If the '-T gif' option is used, a transparent pseudo-GIF may be produced by setting the `TRANSPARENT_COLOR` environment variable to the name of the background color. See Section 2.7 [graph Environment], page 24.

'--bitmap-size *bitmap_size*'
: (String, default "570x570".) Set the size of the graphics display in which the plot will be drawn, in terms of pixels, to be *bitmap_size*. This is relevant only to `graph -T X`, `graph -T pnm`, and `graph -T gif`, for which the graphics display size can be expressed in terms of pixels. The environment variable `BITMAPSIZE` can equally well be used to specify the size.

The graphics display used by `graph -T X` is an X window. If you choose a rectangular (non-square) window size, the fonts in the plot will be scaled anisotropically, i.e., by different factors in the horizontal and vertical direction. This requires an X11R6 display. Any font that cannot be scaled in this way will be replaced by a default scalable font, such as the Hershey vector font "Hershey-Serif".

For backward compatibility, the X resource `Xplot.geometry`, which can be set by the user, may be used to set the window size, instead of '`--bitmap-size`' or `BITMAPSIZE`.

'`--frame-color` *name*'
: (String, default "black".) Set the color used for drawing the plot frame, and for drawing monochrome datasets (if any) to be *name*. An unrecognized name sets the color to the default. For information on what names are recognized, see Appendix B [Color Names], page 146.

'`--frame-line-width` *frame_line_width*'
: (Float, default −1.0.) Set the thickness of lines in the plot frame, as a fraction of the size (i.e., minimum dimension) of the graphics display, to *frame_line_width*. A negative value means that the default value for the line thickness provided by the `libplot` graphics library should be used. This is usually 1/850 times the size of the display, although if '`-T X`', '`-T pnm`', or `-T gif` is specified, it is zero. By convention, a zero-thickness line is the thinnest line that can be drawn. This is the case in all output formats. Note, however, that the drawing editors `idraw` and `xfig` treat zero-thickness lines as invisible.

  `graph -T tek` does not support drawing lines with other than a default thickness, and `graph -T hpgl` does not support doing so if the environment variable `HPGL_VERSION` is set to a value less than "2" (the default).

'`--max-line-length` *max_line_length*'
: (Integer, default 500.) Set the maximum number of points that a polygonal line drawn through any dataset may contain, before it is flushed to the display device, to equal *max_line_length*. If this flushing occurs, the polygonal line will be split into two or more sub-lines, though the splitting should not be noticeable. Splitting will not take place if the '`-q`' option, which requests filling, is used.

  The reason for splitting long polygonal lines is that some display devices (e.g., old Postscript printers and HP-GL pen plotters) have limited buffer sizes. The environment variable `MAX_LINE_LENGTH` can also be used to specify the maximum line length. This option has no effect on `graph -T tek` or raw `graph`, since they draw polylines in real time and have no buffer limitations.

'`--page-size` *pagesize*'
: (String, default "letter".) Set the size of the page on which the plot will be positioned. This is relevant only to `graph -T ai`, `graph -T ps`, `graph -T fig`, `graph -T pcl`, and `graph -T hpgl`. "letter" means an 8.5 in by 11 in page. Any ISO page size in the range "a0"..."a4" or ANSI page size in the range "a"..."e" may be specified ("letter" is an alias for "a" and "tabloid" is an alias for "b").

"legal", "ledger", and "b5" are recognized page sizes also. The environment variable PAGESIZE can equally well be used to specify the page size.

For graph -T ai and graph -T ps, the graphics display within which the plot is drawn will be a square region centered on the specified page, occupying its full width (with allowance being made for margins). For graph -T fig, it will be a square region of the same size, located in the upper left corner of an xfig display. For graph -T pcl and graph -T hpgl, the graphics display will be a square region of the same size, but may be positioned differently. Fine control over its positioning on the page can be accomplished by setting certain environment variables (see Section 2.7 [graph Environment], page 24).

'--pen-colors *colors*'

(String, default "1=red:2=green:3=blue:4=magenta:5=cyan".) Set the colors of the pens used for drawing plots, as numbered, to be *colors*. The format should be self-explanatory. An unrecognized name sets the corresponding color to the default. For information on what names are recognized, see Appendix B [Color Names], page 146.

'--rotation *angle*'

(Integer, default 0.) Set the rotation angle of the plot within the graphics display to be *angle* degrees. Recognized values are 0, 90, 180, and 270. The rotation is counterclockwise.

This option is used for switching between portrait mode and landscape mode. Postmodernists may also find it useful.

'--title-font-name *font_name*'

(String, default "Helvetica" except for graph -T pnm, graph -T gif, graph -T pcl, graph -T hpgl and graph -T tek, for which "HersheySerif" is the default.) Set the font used for the plot title to be *font_name*. Normally the font used for the plot title is the same as that used for labeling the axes and the ticks along the axes, as specified by the '-F' option. But the '--title-font-name' option can be used to override this. Font names are case-insensitive. If the specified font is not available, the default font will be used. Which fonts are available depends on which '-T' option is used. For a list of all fonts, see Section A.1 [Text Fonts], page 124. The plotfont utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

'--title-font-size *size*'

(Float, default 0.07.) Set the size of the font used for the top label ('title'), as specified by the '-L' option, to be *size*. The size is specified as a fraction of the minimum dimension (width or height) of the plotting box.

'--toggle-rotate-y-label'

Position the label on the y axis (which is set with the '-Y' option) horizontally instead of vertically, or vice versa. By default the label is rotated, so that it is parallel to the y axis. But some display devices (e.g., old X Window System displays) cannot handle rotated fonts.

2.6.2 Dataset options

The following options affect the way in which individual datasets are read from files, and drawn as part of a plot. They should appear on the command line before the file containing the datasets whose reading or rendering they will affect. They may appear more than once on a command line, if more than one file is to be read.

The following three options affect the way in which datasets are read from files.

'`-I` *data-format*'
'`--input-format` *data-format*'
: This specifies which format the subsequent input file(s) are in.

  '`a`' ASCII format. Each input file is a sequence of floating point numbers, interpreted as the x and y coordinates of the successive data points in a dataset. The x and y coordinates of a point need not appear on the same line, and points need not appear on different lines. But if a blank line occurs (i.e., two newlines in succession are seen), it is interpreted as the end of a dataset, and the beginning of the next.

  '`e`' ASCII format, including error bars. Similar to '`a`' format, except that triples (x,y,*error*) appear instead of pairs (x, y).

  '`g`' The ASCII 'table' format produced by the `gnuplot` plotting program.

  '`f`' Single precision binary format. Each input file is a sequence of single precision floating point numbers, interpreted as forming pairs (x,y). Successive datasets are separated by a single occurrence of the quantity `FLT_MAX`, which is the largest possible single precision floating point number. On most machines this is approximately 3.4×10^{38}.

  '`d`' Double precision binary format. Each input file is a sequence of double precision floating point numbers, interpreted as forming pairs (x,y). Successive datasets are separated by a single occurrence of the quantity `DBL_MAX`, which is the largest possible double precision floating point number. On most machines this is approximately 1.8×10^{308}.

  '`i`' Integer binary format. Each input file is a sequence of integers, interpreted as forming pairs (x,y). Successive datasets are separated by a single occurrence of the quantity `INT_MAX`, which is the largest possible integer. On most machines this is $2^{31} - 1$.

'`-a [`*step_size* `[`*lower_limit*`]]`'
'`--auto-abscissa [`*step_size* `[`*lower_limit*`]]`'
: (Floats, defaults 1.0 and 0.0.) Automatically generate abscissa (x) values. Irrespective of data format ('`a`', '`e`', '`f`', '`d`', or '`i`'), this option specifies that the abscissa (x) values are missing from the input file: the dataset(s) to be read contain only ordinate (y) values. The increment from each x value to the next

will be *step_size*, and the first x value will be *lower_limit*. To return to reading abscissa values from the input, i.e., for subsequent input files, you would use '-a 0', which disables automatic generation of the abscissa values and returns *step_size* and *lower_limit* to their default values.

'-B'
'--toggle-auto-bump'
: By default the linemode (set with '-m', see below) is 'bumped' (incremented by unity) at the beginning of each new dataset. This option toggles auto-bumping: it turns it off if it was on, and on if it was off.

The following options affect the way in which individual datasets are drawn as part of a plot. These options set the six 'attributes' (symbol type, symbol font, linemode, line thickness, fill fraction, and color/monochrome) that each dataset has.

'-m *line_mode*'
'--line-mode *line_mode*'
: (Integer, default 1.) *line_mode* specifies the mode (i.e., style) of the lines drawn between successive points in a dataset. By convention, linemode #0 means no line at all (points are disconnected). If the dataset is being rendered in monochrome, the interpretation of *line_mode* is as follows.

1. solid
2. dotted
3. dotdashed
4. shortdashed
5. longdashed

Thereafter (i.e., for *line_mode* greater than 5) the sequence of five linemodes repeats. So besides linemode #0, there are a total of five distinct monochrome linemodes. If the dataset is being rendered in color (as may be requested with the '-C' option), the interpretation of linemodes #1 through #5 is instead

1. red, solid
2. green, solid
3. blue, solid
4. magenta, solid
5. cyan, solid

Linemodes #6 through #10 use the same five colors, but are dotted; linemodes #11 through #15 are dotdashed; linemodes #16 through #20 are shortdashed; and linemodes #21 through #25 are longdashed. So besides linemode #0, there are a total of 25 distinct colored linemodes. A negative linemode indicates that no line should be drawn, but that the plotting symbol, if any (see below), should be in the color of the corresponding positive linemode.

'-S [*symbol_number* [*symbol_size*]]'
'--symbol [*symbol_number* [*symbol_size*]]'
: (Integer and float, defaults 0 and 0.03.) Draw a marker symbol at each data point. *symbol_number* specifies the symbol type, and *symbol_size* specifies the

font size of the symbol, as a fraction of the minimum dimension (width or height) of the plotting box. If the dataset is being rendered in color, the symbol will have the color of the line that is being drawn to connect the data points. You may request that symbols be drawn without any line connecting them by using the '-m' option to specify a negative linemode (see above). The following table lists the first few marker symbols (by convention, symbol #0 means no symbol at all).

1. dot ($\cdot$)
2. plus (+)
3. asterisk ($*$)
4. circle ($\circ$)
5. cross ($\times$)

Marker symbols 0 through 31 are furnished by the `libplot` graphics library. See Section A.5 [Marker Symbols], page 144. Symbol numbers greater than or equal to 32 are interpreted as characters to be selected from a symbol font, which can be set with the '`--symbol-font-name`' option (see below).

'`-W` *line_width*'
'`--line-width` *line_width*'

(Float, default −1.0.) Set the thickness of the lines used to join successive points in a dataset, as a fraction of the size (i.e., minimum dimension) of the graphics display, to *line_width*. A negative value means that the default value for the line thickness provided by the `libplot` graphics library should be used. This is usually 1/850 times the size of the display, although if '`-T X`', '`-T pnm`', or `-T gif` is specified, it is zero. By convention, a zero-thickness line is the thinnest line that can be drawn. This is the case in all output formats. Note, however, that the drawing editors `idraw` and `xfig` treat zero-thickness lines as invisible.

`graph -T tek` does not support drawing lines with other than a default thickness, and `graph -T hpgl` does not support doing so if the environment variable `HPGL_VERSION` is set to a value less than "2" (the default).

'`-q` *fill_fraction*'
'`--fill-fraction` *fill_fraction*'

(Float, default −1.0.) If successive points in a dataset are joined by line segments, set the shading intensity for the polygon formed by the line segments to be *fill_fraction*. A solid polygon (i.e., one filled with the 'pen color' used for drawing the line segments) is obtained by choosing *fill_fraction*=1.0. The interior of the polygon will be white if *fill_fraction*=0.0. The polygon will be unfilled (transparent) if *fill_fraction* is negative.

If the polygon intersects itself, the 'even-odd fill rule' will normally be used to determine which points are inside rather than outside, i.e., to determine which portions of the polygon should be shaded. The even-odd fill rule is explained in the *Postscript Language Reference Manual*.

The '`-q`' option has no effect in `graph -T tek`, and it is only partly effective in `graph -T hpgl` if the environment variable `HPGL_VERSION` is set to "1".

'`-C`'
'`--toggle-use-color`'
: Toggle between color and monochrome rendering of datasets. The interpretation of linemode depends on whether the rendering is being performed in color or monochrome; see the '`-m`' option above.

'`--symbol-font-name` *symbol_font_name*'
: (String, default "ZapfDingbats" unless '`-T pnm`', '`-T gif`', '`-T pcl`', '`-T hpgl`', or `-T tek` is specified, in which case it is "HersheySerif".) Set the symbol font, from which plotting symbols numbered 32 and higher are selected, to be *symbol_font_name*. Font names are case-insensitive. If the specified font is not available, the default font will be used. Which fonts are available depends on which '`-T`' option is used. For example, if the '`-T pcl`' or '`-T hpgl`' option is used then normally the Wingdings font, which is an alternative source of symbols, becomes available. For a list of all fonts, see Section A.1 [Text Fonts], page 124. The `plotfont` utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

2.6.3 Multiplot options

The following options are used for multiplotting (placing several plots on a display, or a page, at once).

'`--reposition` *x y size*'
: (Floats, defaults 0.0, 0.0, 1.0) Set the 'virtual display' within which the next plot will be drawn to be a square of size *size*, with lower left corner (*x*,*y*). Normalized coordinates are used here: (0,0) means the lower left corner of the physical display and (1,1) means the upper right corner of the physical display. The size of the plot within the virtual display may be adjusted with the '`-h`' and '`-w`' options, and its position within the virtual display with the '`-u`' and '`-w`' options. After a '`--reposition`' command, the arguments of those four options will be interpreted in terms of the virtual display, not the physical display.

'`--blankout` *blankout_fraction*'
: (Float, default 1.3.) When drawing each additional plot of a multiplot, it is desirable to clear the region of the display on which the plot will be drawn. If *blankout_fraction*=1.0, the region covered by the plot's plotting box will be cleared. If *blankout_fraction*=1.3, a region 30% larger in each dimension will be cleared. This is appropriate for inset plots; 1.0 would be appropriate for side by side plots. Note that `graph -T tek` cannot clear regions, and `graph -T hpgl` cannot clear them if the environment variables `HPGL_VERSION` and `HPGL_OPAQUE_MODE` are set to non-default values (i.e., values other than "2" and "yes", respectively).

2.6.4 Raw `graph` options

The following option is relevant only to raw `graph`, i.e., is relevant only if no display type or output format is specified with the '`-T`' option. In this case `graph` outputs a graphics metafile, which may be translated to other formats by invoking `plot`. This option should appear on the command line before any file names, since it affects the output of the plot (or multiplot) as a whole.

'-O'
'--portable-output'
: Output the portable (human-readable) version of GNU metafile format, rather than a binary version (the default). This can also be requested by setting the environment variable `META_PORTABLE` to "yes".

2.6.5 Informational options

The following options request information.

'--help' Print a list of command-line options, and then exit.

'--help-fonts'
: Print a table of available fonts, and then exit. The table will depend on which display type or output format is specified with the '-T' option. `graph -T X`, `graph -T ai`, `graph -T ps`, and `graph -T fig` each support the 35 standard Postscript fonts. `graph -T ai`, `graph -T pcl`, and `graph -T hpgl` support the 45 standard PCL 5 fonts, and `graph -T pcl` and `graph -T hpgl` support a number of Hewlett–Packard vector fonts. All of the preceding, together with `graph -T pnm`, `graph -T gif`, and `graph -T tek`, support a set of 22 Hershey vector fonts. Raw `graph` in principle supports any of these fonts, since its output must be translated to other formats with `plot`. The `plotfont` utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

'--list-fonts'
: Like '--help-fonts', but lists the fonts in a single column to facilitate piping to other programs. If no display type or output format is specified with the '-T' option, the full set of supported fonts is listed.

'--version'
: Print the version number of `graph` and the plotting utilities package, and exit.

2.7 Environment variables

The behavior of `graph` is affected by several environment variables. We have already mentioned the environment variables `BITMAPSIZE`, `PAGESIZE`, `BG_COLOR`, and `MAX_LINE_LENGTH`. They serve as backups for the options '--bitmap-size', '--page-size', '--bg-color', and '--max-line-length'. The remaining environment variables are specific to individual output formats.

`graph -T X`, which pops up a window on an X Window System display and draws graphics in it, checks the `DISPLAY` environment variable. The value of this variable determines the display on which the window will be popped up.

`graph -T pnm`, which produces output in Portable Anymap (PBM/PGM/PPM) format, is affected by the `PNM_PORTABLE` environment variable. If its value is "yes", the output file will be in the portable (human readable) version of PBM, PGM, or PPM format, rather than the default (binary) version.

`graph -T gif`, which produces output in pseudo-GIF format, is affected by two environment variables. If the value of the `INTERLACE` variable is "yes", the pseudo-GIF output file will be in interlaced format. Also, if the value of the `TRANSPARENT_COLOR` environment variable is the name of a color that appears in the output file, that color will be treated as transparent by most applications that read GIF files. For information on what color names are recognized, see Appendix B [Color Names], page 146.

`graph -T pcl`, which produces PCL 5 output for Hewlett–Packard printers and plotters, is affected by several environment variables. The position of the graphics display on the page can be adjusted by setting the `PCL_XOFFSET` and `PCL_YOFFSET` environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the `PCL_ROTATE` environment variable to "yes". This is not the same as the rotation obtained with the `--rotation` option, which sets the rotation angle of the plot within the display. Besides "no" and "yes", recognized values for the `PCL_ROTATE` variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively.

The variable `PCL_ASSIGN_COLORS` is also recognized. It should be set to "yes" when producing PCL 5 output for a color printer or other color device. This will ensure accurate color reproduction by giving the output device complete freedom in assigning colors, internally, to its “logical pens”. If it is "no" then the device will use a fixed set of colored pens, and will emulate other colors by shading. The default is "no" because monochrome PCL 5 devices, which are much more common than colored ones, must use shading to emulate color.

`graph -T hpgl`, which produces Hewlett–Packard Graphics Language output, is also affected by several environment variables. The most important is `HPGL_VERSION`, which may be set to "1", "1.5", or "2" (the default). "1" means that the output should be generic HP-GL, "1.5" means that the output should be suitable for the HP7550A graphics plotter and the HP758x, HP7595A and HP7596A drafting plotters (HP-GL with some HP-GL/2 extensions), and "2" means that the output should be modern HP-GL/2. If the version is "1" or "1.5" then the only available fonts will be vector fonts, and all lines will be drawn with a default thickness (the ‘`-W`’ option will not work). Additionally, if the version is "1" then the filling of arbitrary curves with solid color will not be supported (the ‘`-q`’ option may be used to fill circles and rectangles aligned with the coordinate axes, though).

The position of the `graph -T hpgl` graphics display on the page can be adjusted by setting the `HPGL_XOFFSET` and `HPGL_YOFFSET` environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the `HPGL_ROTATE` environment variable to "yes". This is not the same as the rotation obtained with the `--rotation` option, which sets the rotation angle of the plot within the display. Besides "no" and "yes", recognized values for the `HPGL_ROTATE` variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively. "180" and "270" are supported only if `HPGL_VERSION` is "2" (the default).

Opaque filling and the drawing of visible white lines are supported only if `HPGL_VERSION` is "2" (the default) and the environment variable `HPGL_OPAQUE_MODE` is "yes" (the default). If the value is "no" then opaque filling will not be used, and white lines (if any), which are normally drawn with pen #0, will not be drawn. This feature is to accommodate older HP-GL/2 devices. HP-GL/2 pen plotters, for example, do not support opacity or the use of pen #0 to draw visible white lines. Some older HP-GL/2 devices reportedly malfunction if asked to draw opaque objects.

By default, `graph -T hpgl` will draw with a fixed set of pens. Which pens are present may be specified by setting the `HPGL_PENS` environment variable. If `HPGL_VERSION` is "1", the default value of `HPGL_PENS` is "1=black"; if `HPGL_VERSION` is "1.5" or "2", the default value of `HPGL_PENS` is "1=black:2=red:3=green:4=yellow:5=blue:6=magenta:7=cyan". The format should be self-explanatory. By setting `HPGL_PENS`, you may specify a color for any pen in the range #1...#31. For information on what color names are recognized, see Appendix B [Color Names], page 146. Pen #1 must always be present, though it need not be black. Any other pen in the range #1...#31 may be omitted.

If `HPGL_VERSION` is "2" then `graph -T hpgl` will also be affected by the environment variable `HPGL_ASSIGN_COLORS`. If the value of this variable is "yes", then `graph -T hpgl` will not be restricted to the palette specified in `HPGL_PENS`: it will assign colors to “logical pens” in the range #1...#31, as needed. The default value is "no" because other than color LaserJet printers and DesignJet plotters, not many HP-GL/2 devices allow the assignment of colors to logical pens.

`graph -T tek`, which produces output for a Tektronix terminal or emulator, checks the `TERM` environment variable. If the value of `TERM` is "xterm", "xterms", or "kterm", it is taken as a sign that the current application is running in an X Window System VT100 terminal emulator: an `xterm`. Before drawing graphics, `graph -T tek` will emit an escape sequence that causes the terminal emulator’s auxiliary Tektronix window, which is normally hidden, to pop up. After the graphics are drawn, an escape sequence that returns control to the original VT100 window will be emitted. The Tektronix window will remain on the screen.

If the value of `TERM` is "kermit", "ansi.sys", "ansissys", "ansi.sysk", or "ansisysk", it is taken as a sign that the current application is running in the VT100 terminal emulator provided by the MS-DOS version of `kermit`. Before drawing graphics, `graph -T tek` will emit an escape sequence that switches the terminal emulator to Tektronix mode. Also, some of the Tektronix control codes emitted by `graph -T tek` will be `kermit`-specific. There will be a limited amount of color support, which is not normally the case (the 16 `ansi.sys` colors will be supported). After drawing graphics, `graph -T tek` will emit an escape sequence that returns the emulator to VT100 mode. The key sequence ‘ALT minus’ can be employed manually within `kermit` to switch between the two modes.

3 The plot Program

3.1 How to use plot

The GNU plot filter `plot` displays GNU graphics metafiles or translates them to other formats. It will take input from files specified on the command line or from standard input. The '-T' option is used to specify the desired output format. Supported output formats include "X", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", and "meta" (the default).

The metafile format is a device-independent format for storage of vector graphics. By default, it is a binary rather than a human-readable format (see Appendix C [Metafiles], page 147). Each of the `graph`, `pic2plot`, `tek2plot`, and `plotfont` utilities will write a graphics metafile to standard output if no '-T' option is specified on its command line. The `libplot` graphics library may also be used to produce metafiles. Metafiles may contain arbitrarily many pages of graphics, but each metafile produced by `graph` contains only a single page.

`plot`, like the metafile format itself, is useful if you wish to preserve a vector graphics file, and display or edit it with more than one drawing editor. The following example shows how you may do this.

To produce a plot of data arranged as alternating x and y coordinates in an ASCII file, you may use `graph` as follows:

```
graph < ascii_data_file > test.meta
```

The file 'test.meta' will be a single-page graphics metafile. Similarly, to create in metafile format a plot consisting of a simple figure, you may do:

```
echo 0 0 1 1 2 0 | spline | graph > test.meta
```

To display any such plot on an X Window System display, you would do

```
plot -T X test.meta
```

or

```
plot -T X < test.meta
```

To print the plot on a Postscript printer, you would do something like

```
plot -T ps < test.meta | lpr
```

To edit it with the free `idraw` drawing editor, you would do

```
plot -T ps < test.meta > test.ps
idraw test.ps
```

To produce a "portable anymap" (a file in PBM, PGM, or PPM format, whichever is most appropriate) you would do

```
plot -T pnm < test.meta > test.pnm
```

and to produce a pseudo-GIF file, you would do

```
plot -T gif < test.meta > test.gif
```

Similarly, to produce a version of the plot that can be edited with Adobe Illustrator, you would do

```
plot -T ai < test.meta > test.ai
```

and to produce a version that can be edited with the free `xfig` drawing editor, you would do

```
plot -T fig < test.meta > test.fig
xfig test.fig
```

Other formats may be obtained by using `plot -T pcl`, `plot -T hpgl`, and `plot -T tek`.

`plot` may behave differently depending on the environment in which it is invoked. In particular, `plot -T ai`, `plot -T ps`, `plot -T fig`, `plot -T pcl`, and `plot -T hpgl` are affected by the environment variable `PAGESIZE`. `plot -T X`, `plot -T pnm`, and `plot -T gif` are affected by the environment variable `BITMAPSIZE`. The `DISPLAY` environment variable affects the operation of `plot -T X`, and the `TERM` environment variable affects the operation of `plot -T tek`. There are also several environment variables that affect the operation of `plot -T pcl` and `plot -T hpgl`. For a complete discussion of the effects of the environment on `plot`, see Section 3.3 [plot Environment], page 32.

3.2 `plot` command-line options

The plot filter `plot` translates GNU graphics metafiles to other formats. The '`-T`' option is used to specify the output format or display type. Files in metafile format are produced by GNU `graph`, `pic2plot`, `tek2plot`, `plotfont`, and other applications that use the GNU `libplot` graphics library. For technical details on the metafile format, see Appendix C [Metafiles], page 147.

Input file names may be specified anywhere on the command line. That is, the relative order of file names and command-line options does not matter. If no file names are specified, or the file name '`-`' is specified, the standard input is read. The output file is written to standard output, unless the '`-T X`' option is specified. In that case the output is displayed in a window or windows on an X Window System display, and there is no output file.

The full set of command-line options is listed below. There are four sorts of option:

1. Options setting the values of drawing parameters.
2. Options relevant only to raw `plot`, i.e., relevant only if no display type or output format is specified with the '`-T`' option.
3. Options specifying the type of metafile format the input is in (for backward compatibility only).
4. Options requesting information (e.g., '`--help`').

Each option that takes an argument is followed, in parentheses, by the type and default value of the argument.

The following options set the values of drawing parameters.

'`-T` *type*'
'`--display-type` *type*'

(String, default "meta".) Select a display type or output format of type *type*, which may be one of the strings "X", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", and "meta". These refer respectively to the X Window System, portable anymap (PBM/PGM/PPM) format, pseudo-GIF format, the format used by Adobe Illustrator, `idraw`-editable Postscript, the format used by the `xfig` drawing editor, the Hewlett–Packard PCL 5 printer language, the Hewlett–Packard Graphics Language (by default, HP-GL/2), Tektronix format, and device-independent GNU graphics metafile format.

'`-p` *n*'
'`--page-number` *n*'
: (Positive integer.) Display only page number *n*, within the metafile or sequence of metafiles that is being translated.

  Metafiles may consist of one or more pages, numbered beginning with 1. Also, each page may contain multiple 'frames'. `plot -T X` or `plot -T tek`, which plot in real time, will separate successive frames by screen erasures. `plot -T pnm`, `plot -T gif`, `plot -T ai`, `plot -T ps`, `plot -T fig`, `plot -T pcl`, and `plot -T hpgl`, which do not plot in real time, will display only the last frame of any multi-frame page.

  The default behavior, if '`-p`' is not used, is to display all pages. For example, `plot -T X` displays each page in its own X window. If the '`-T pnm`' option, the '`-T gif`' option, the '`-T ai`' option, or the '`-T fig`' option is used, the default behavior is to display only the first page, since files in PNM, pseudo-GIF, AI, or Fig format may contain only a single page of graphics.

  Most metafiles produced by the GNU plotting utilities (e.g., by raw `graph`) contain only a single page, consisting of two frames: an empty one to clear the display, and a second one containing graphics.

'`-s`'
'`--merge-pages`'
: Merge all displayed pages into a single page, and also merge all 'frames' within each displayed page.

  This option is useful when merging together single-page plots from different sources. For example, it can be used to merge together plots obtained from separate invocations of `graph`. This is an alternative form of multiplotting (see Section 2.4 [Multiplotting], page 11).

'`--bitmap-size` *bitmap_size*'
: (String, default "570x570".) Set the size of the graphics display in which the plot will be drawn, in terms of pixels, to be *bitmap_size*. This is relevant only to `plot -T X`, `plot -T pnm`, and `plot -T gif`, for which the graphics display size can be expressed in terms of pixels. The environment variable `BITMAPSIZE` can equally well be used to specify the size.

  For backward compatibility, the X resource `Xplot.geometry`, which can be set by the user, may be used to set the window size, instead of '`--bitmap-size`' or `BITMAPSIZE`.

'`--max-line-length` *max_line_length*'
: (Integer, default 500.) Set the maximum number of points that a polygonal line may contain, before it is flushed to the display device, to equal *max_line_length*.

If this flushing occurs, the polygonal line will be split into two or more sub-lines, though the splitting should not be noticeable. Splitting will not take place if the line is the boundary of a filled polygon.

The reason for splitting long polygonal lines is that some display devices (e.g., old Postscript printers and HP-GL pen plotters) have limited buffer sizes. The environment variable `MAX_LINE_LENGTH` can also be used to specify the maximum line length. This option has no effect on `plot -T tek` or raw `plot`, since they draw polylines in real time and have no buffer limitations.

'`--page-size` *pagesize*'

(String, default "letter".) Set the size of the page on which the plot will be positioned. This is relevant only to `plot -T ai`, `plot -T ps`, `plot -T fig`, `plot -T pcl`, and `plot -T hpgl`. "letter" means an 8.5 in by 11 in page. Any ISO page size in the range "a0"..."a4" or ANSI page size in the range "a"..."e" may be specified ("letter" is an alias for "a" and "tabloid" is an alias for "b"). "legal", "ledger", and "b5" are recognized page sizes also. The environment variable `PAGESIZE` can equally well be used to specify the page size.

For `plot -T ai` and `plot -T ps`, the graphics display within which the plot is drawn will be a square region centered on the specified page, occupying its full width (with allowance being made for margins). For `plot -T fig`, it will be a square region located of the same size, located in the upper left corner of an `xfig` display. For `plot -T pcl` and `plot -T hpgl`, the graphics display will be a square region of the same size, but may be positioned differently. Fine control over its positioning on the page may be accomplished by setting certain environment variables (see Section 3.3 [plot Environment], page 32).

The following options set the initial values of additional drawing parameters. All of these may be overridden by directives in the metafile itself. In fact, these options are useful mostly for plotting old metafiles in the pre-GNU 'plot(5)' format, which did not include such directives.

'`--bg-color` *name*'

(String, default "white".) Set the color used for the plot background to be *name*. This is relevant only to `plot -T X`, `plot -T pnm`, and `plot -T gif`. An unrecognized name sets the color to the default. For information on what names are recognized, see Appendix B [Color Names], page 146. The environment variable `BG_COLOR` can equally well be used to specify the background color.

If the '`-T gif`' option is used, a transparent pseudo-GIF may be produced by setting the `TRANSPARENT_COLOR` environment variable to the name of the background color. See Section 3.3 [plot Environment], page 32.

'`-f` *font_size*'
'`--font-size` *font_size*'

(Float, initial value device-dependent.) Set the initial size of the font used for rendering text, as a fraction of the width of the graphics display, to *font_size*.

'-F *font_name*'
'--font-name *font_name*'
(String, default "Helvetica" except for plot -T pnm, plot -T gif, plot -T pcl, plot -T hpgl, plot -T tek, and raw plot, for all of which "HersheySerif" is the default.) Set the font initially used for text (i.e., for 'labels') to *font_name*. Font names are case-insensitive. If the specified font is not available, the default font will be used. Which fonts are available depends on which '-T' option is used. For a list of all fonts, see Section A.1 [Text Fonts], page 124. The plotfont utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

'-W *line_width*'
'--line-width *line_width*'
(Float, default −1.0.) Set the thickness of lines, as a fraction of the size (i.e., minimum dimension) of the graphics display, to *line_width*. A negative value means that the default value provided by the libplot graphics library should be used. This is usually 1/850 times the size of the display, although if '-T X', '-T pnm', or -T gif is specified, it is zero. By convention, a zero-thickness line is the thinnest line that can be drawn. This is the case in all output formats. Note, however, that the drawing editors idraw and xfig treat zero-thickness lines as invisible.

plot -T tek does not support drawing lines with other than a default thickness, and plot -T hpgl does not support doing so if the environment variable HPGL_VERSION is set to a value less than "2" (the default).

'--pen-color *name*'
(String, default "black".) Set the pen color to be *name*. An unrecognized name sets the pen color to the default. For information on what color names are recognized, see Appendix B [Color Names], page 146.

The following option is relevant only to raw plot, i.e., relevant only if no output type is specified with the '-T' option. In this case plot outputs a graphics metafile, which may be translated to other formats by a second invocation of plot.

'-O'
'--portable-output'
Output the portable (human-readable) version of GNU metafile format, rather than a binary version (the default). This can also be requested by setting the environment variable META_PORTABLE to "yes".

plot will automatically determine which type of GNU metafile format the input is in. There are two types: binary (the default) and portable (human-readable). The binary format is machine-dependent. See Appendix C [Metafiles], page 147.

For compatibility with older plotting software, the reading of input files in the pre-GNU 'plot(5)' format is also supported. This is normally a binary format, with each integer in the metafile represented as a pair of bytes. The order of the two bytes is machine dependent. You may specify that input file(s) are in plot(5) format rather than ordinary GNU metafile format by using either the '-h' option ("high byte first") or the '-l' option ("low byte first"), whichever is appropriate. Some non-GNU systems support an ASCII (human-readable) variant of plot(5) format. You may specify that the input is in this format by using the '-A' option. Irrespective of the variant, a file in plot(5) format includes only one page of graphics.

'-h'
'--high-byte-first-input'
: Input file(s) are assumed to be in traditional 'plot(5)' metafile format, with the high-order byte of each integer occurring first.

'-l'
'--low-byte-first-input'
: Input file(s) are assumed to be in traditional 'plot(5)' metafile format, with the low-order byte of each integer occurring first.

'-A'
'--ascii-input'
: Input file(s) are assumed to be in the ASCII variant of traditional 'plot(5)' metafile format.

The following options request information.

'--help' Print a list of command-line options, and then exit.

'--help-fonts'
: Print a table of available fonts, and then exit. The table will depend on which display type or output format is specified with the '-T' option. `plot -T X`, `plot -T ai`, `plot -T ps`, and `plot -T fig` each support the 35 standard Postscript fonts. `plot -T ai`, `plot -T pcl`, and `plot -T hpgl` support the 45 standard PCL 5 fonts, and `plot -T pcl` and `plot -T hpgl` support a number of Hewlett–Packard vector fonts. All of the preceding, together with `plot -T pnm`, `plot -T gif`, and `plot -T tek`, support a set of 22 Hershey vector fonts. Raw `plot` in principle supports any of these fonts, since its output must be translated to other formats with `plot`. The `plotfont` utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

'--list-fonts'
: Like '--help-fonts', but lists the fonts in a single column to facilitate piping to other programs. If no display type or output format is specified with the '-T' option, the full set of supported fonts is listed.

'--version'
: Print the version number of `plot` and the plotting utilities package, and exit.

3.3 Environment variables

The behavior of `plot` is affected by several environment variables. We have already mentioned the environment variables `BITMAPSIZE`, `PAGESIZE`, `BG_COLOR`, and `MAX_LINE_LENGTH`. They serve as backups for the options '--bitmap-size', '--page-size', '--bg-color', and '--max-line-length'. The remaining environment variables are specific to individual output formats.

plot -T X, which pops up a window on an X Window System display and draws graphics in it, checks the DISPLAY environment variable. The value of this variable determines the display on which the window will be popped up.

plot -T pnm, which produces output in Portable Anymap (PBM/PGM/PPM) format, is affected by the PNM_PORTABLE environment variable. If its value is "yes", the output file will be in the portable (human readable) version of PBM, PGM, or PPM format, rather than the default (binary) version.

plot -T gif, which produces output in pseudo-GIF format, is affected by two environment variables. If the value of the INTERLACE variable is "yes", the pseudo-GIF output file will be in interlaced format. Also, if the value of the TRANSPARENT_COLOR environment variable is the name of a color that appears in the output file, that color will be treated as transparent by most applications that read GIF files. For information on what color names are recognized, see Appendix B [Color Names], page 146.

plot -T pcl, which produces PCL 5 output for Hewlett–Packard printers and plotters, is affected by several environment variables. The position of the graphics display on the page can be adjusted by setting the PCL_XOFFSET and PCL_YOFFSET environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the PCL_ROTATE environment variable to "yes". Besides "no" and "yes", recognized values for this variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively.

The variable PCL_ASSIGN_COLORS is also recognized. It should be set to "yes" when producing PCL 5 output for a color printer or other color device. This will ensure accurate color reproduction by giving the output device complete freedom in assigning colors, internally, to its "logical pens". If it is "no" then the device will use a fixed set of colored pens, and will emulate other colors by shading. The default is "no" because monochrome PCL 5 devices, which are much more common than colored ones, must use shading to emulate color.

plot -T hpgl, which produces Hewlett–Packard Graphics Language output, is also affected by several environment variables. The most important is HPGL_VERSION, which may be set to "1", "1.5", or "2" (the default). "1" means that the output should be generic HP-GL, "1.5" means that the output should be suitable for the HP7550A graphics plotter and the HP758x, HP7595A and HP7596A drafting plotters (HP-GL with some HP-GL/2 extensions), and "2" means that the output should be modern HP-GL/2. If the version is "1" or "1.5" then the only available fonts will be vector fonts, and all lines will be drawn with a default thickness (the '-W' option will not work). Additionally, if the version is "1" then the filling of arbitrary curves with solid color will not be supported (circles and rectangles aligned with the coordinate axes may be filled, though).

The position of the plot -T hpgl graphics display on the page can be adjusted by setting the HPGL_XOFFSET and HPGL_YOFFSET environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the HPGL_ROTATE environment variable to "yes". Besides "no" and "yes", recognized values for this variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to

"0" and "90", respectively. "180" and "270" are supported only if HPGL_VERSION is "2" (the default).

Opaque filling and the drawing of visible white lines are supported only if HPGL_VERSION is "2" (the default) and the environment variable HPGL_OPAQUE_MODE is "yes" (the default). If the value is "no" then opaque filling will not be used, and white lines (if any), which are normally drawn with pen #0, will not be drawn. This feature is to accommodate older HP-GL/2 devices. HP-GL/2 pen plotters, for example, do not support opacity or the use of pen #0 to draw visible white lines. Some older HP-GL/2 devices reportedly malfunction if asked to draw opaque objects.

By default, `plot -T hpgl` will draw with a fixed set of pens. Which pens are present may be specified by setting the HPGL_PENS environment variable. If HPGL_VERSION is "1", the default value of HPGL_PENS is "1=black"; if HPGL_VERSION is "1.5" or "2", the default value of HPGL_PENS is "1=black:2=red:3=green:4=yellow:5=blue:6=magenta:7=cyan". The format should be self-explanatory. By setting HPGL_PENS, you may specify a color for any pen in the range #1...#31. For information on what color names are recognized, see Appendix B [Color Names], page 146. Pen #1 must always be present, though it need not be black. Any other pen in the range #1...#31 may be omitted.

If HPGL_VERSION is "2" then `plot -T hpgl` will also be affected by the environment variable HPGL_ASSIGN_COLORS. If the value of this variable is "yes", then `plot -T hpgl` will not be restricted to the palette specified in HPGL_PENS: it will assign colors to "logical pens" in the range #1...#31, as needed. The default value is "no" because other than color LaserJet printers and DesignJet plotters, not many HP-GL/2 devices allow the assignment of colors to logical pens.

`plot -T tek`, which produces output for a Tektronix terminal or emulator, checks the TERM environment variable. If the value of TERM is "xterm", "xterms", or "kterm", it is taken as a sign that the current application is running in an X Window System VT100 terminal emulator: an `xterm`. Before drawing graphics, `plot -T tek` will emit an escape sequence that causes the terminal emulator's auxiliary Tektronix window, which is normally hidden, to pop up. After the graphics are drawn, an escape sequence that returns control to the original VT100 window will be emitted. The Tektronix window will remain on the screen.

If the value of TERM is "kermit", "ansi.sys", "ansissys", "ansi.sysk", or "ansisysk", it is taken as a sign that the current application is running in the VT100 terminal emulator provided by the MS-DOS version of `kermit`. Before drawing graphics, `plot -T tek` will emit an escape sequence that switches the terminal emulator to Tektronix mode. Also, some of the Tektronix control codes emitted by `plot -T tek` will be `kermit`-specific. There will be a limited amount of color support, which is not normally the case (the 16 `ansi.sys` colors will be supported). After drawing graphics, `plot -T tek` will emit an escape sequence that returns the emulator to VT100 mode. The key sequence 'ALT minus' can be employed manually within `kermit` to switch between the two modes.

4 The pic2plot Program

4.1 What pic2plot is used for

The pic2plot program takes one or more files in the pic language, and either displays the figures that they contain on an X Window System display, or produces an output file containing the figures. Many graphics file formats are supported.

The pic language is a 'little language' that was developed at Bell Laboratories for creating box-and-arrow diagrams of the kind frequently found in technical papers and textbooks. A directory containing documentation on the pic language is distributed along with the plotting utilities. On most systems it is installed as '/usr/share/pic2plot' or '/usr/local/share/pic2plot'. The directory includes Brian Kernighan's original technical report on the language, Eric Raymond's tutorial on the GNU implementation, and some sample pic macros contributed by W. Richard Stevens.

The pic language was originally designed to work with the troff document formatter. In that context it is read by a translator called pic, or its GNU counterpart gpic. Since extensive documentation on pic and gpic is available, this section simply gives an example of an input file, and mentions some extra features supported by pic2plot.

A pic file contains one or more figures, each of the box-and-arrow type. Each figure is begun by a line reading .PS, and ended by a line reading .PE. Lines that are not contained in a .PS....PE pair are ignored. Each figure is built from geometrical objects, such as rectangular boxes, circles, ellipses, quarter circles ("arcs"), polygonal lines, and splines. Arcs, polygonal lines, and spline may be equipped with arrowheads. Any object may be labeled with text.

Objects are usually positioned not by specifying their positions in absolute coordinates, but rather by specifying their positions relative to other, previously drawn objects. The following figure is an example.

```
.PS
box "START"; arrow; circle dashed filled; arrow
circle diam 2 thickness 3 "This is a" "big, thick" "circle" dashed; up
arrow from top of last circle; ellipse "loopback" dashed
arrow dotted from left of last ellipse to top of last box
arc cw radius 1/2 from top of last ellipse; arrow
box "END"
.PE
```

If you put this example in a file and run 'pic2plot -T X' on the file, a window containing the figure will be popped up on your X display. Similarly, if you run 'pic2plot -T ps' on the file, a Postscript file containing the figure will be written to standard output. The Postscript file may be edited with the idraw drawing editor. Other graphics formats such as PNM format, pseudo-GIF format, or Fig format (which is editable with the xfig drawing editor) may be obtained similarly. You would use the options '-T pnm', '-T gif', and '-T fig', respectively.

The above example illustrates some of the features of the pic language. By default, successive objects are drawn so as to touch each other. The drawing proceeds in a certain direction, which by default is left-to-right. The 'up' command changes this direction to bottom-to-top, so that the next object (the arrow extending from the top of the big circle) will point upward rather than to the right.

Objects have sizes and other attributes, which may be set globally, or specified on a per-object basis. For example, the diameter of a circle may be specified, or the radius of an arc. An arc may be oriented clockwise rather than counterclockwise by specifying the 'cw' attribute. The line style of most objects may be altered by specifying the 'dashed' or 'dotted' attribute. Also, any object may be labeled, by specifying one or more text strings as attributes. A text string may contain escape sequences that shift the font, append subscripts or superscripts, or include non-ASCII characters and mathematical symbols. See Section A.4 [Text String Format], page 131.

Most sizes and positions are expressed in terms of 'virtual inches'. The use of virtual inches is peculiar to pic2plot. The graphics display used by pic2plot, i.e., its drawing region, is defined to be a square, 8 virtual inches wide and 8 virtual inches high. If the page size for the output file is the "letter" size, which is the default for Postscript output, virtual inches will the same as real inches. But a different page size may be specified; for example, by using the '--page-size a4' option. If so, a virtual inch will simply equal one-eighth of the width of the graphics display.

By default, each figure is centered in the graphics display. You may turn off centering, so that you can use absolute coordinates, by using the '-n' option. For example, a figure consisting only of the object 'arrow from (8,8) to (4,4)' will be positioned in the absence of centering so that the tip of the arrow is at the center of the display. Its tail will be at the upper right corner.

The thickness of lines is not specified in terms of virtual inches. For compatibility with gpic, it is measured in terms of virtual points. The example above, which specifies the 'thickness' attribute of one of the objects, illustrates this. There are 72 virtual points per virtual inch.

If there is more than one figure to be displayed, they will appear in different X windows, or on successive pages of the output file. Some output formats (such as PNM, pseudo-GIF, Illustrator, and Fig) support only a single page of graphics. If any of those output formats is chosen, only the first figure will appear in the output file. Currently, pic2plot cannot produce animated pseudo-GIFs.

The preceding survey does not do justice to the pic language, which is actually a full-featured programming language, with support for variables, looping constructs, etc. Its advanced features make the drawing of large, repetitive diagrams quite easy.

4.2 pic2plot command-line options

The pic2plot program translates files in the pic language, which is used for creating box-and-arrow diagrams of the kind frequently found in technical papers and textbooks, to other graphics formats. The output format or display type is specified with the '-T' option. The possible output formats are the same ten formats that are supported by the GNU graph and plot programs.

Input file names may be specified anywhere on the command line. That is, the relative order of file names and command-line options does not matter. If no file names are specified, or the file name '-' is specified, the standard input is read. The output file is written to standard output, unless the '-T X' option is specified. In that case the output is displayed in one or more windows on an X Window System display, and there is no output file.

The full set of command-line options is listed below. There are three sorts of option:

1. General options.
2. Options relevant only to raw pic2plot, i.e., relevant only if no display type or output format is specified with the '-T' option.
3. Options requesting information (e.g., '--help').

Each option that takes an argument is followed, in parentheses, by the type and default value of the argument.

The following are general options.

'-T *type*'
'--display-type *type*'
: (String, default "meta".) Select a display type or output format of type *type*, which may be one of the strings "X", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", and "meta". These refer respectively to the X Window System, portable anymap (PBM/PGM/PPM) format, pseudo-GIF format, the format used by Adobe Illustrator, idraw-editable Postscript, the format used by the xfig drawing editor, the Hewlett–Packard PCL 5 printer language, the Hewlett–Packard Graphics Language (by default, HP-GL/2), Tektronix format, and device-independent GNU graphics metafile format.

'-d'
'--precision-dashing'
: Draw dashed and dotted lines carefully, i.e., draw each dash and dot as a separate object, so that they may be positioned individually. The default is to use the support for dashed and dotted lines provided by the underlying graphics library, libplot.

  This option may produce slightly better-looking dashed and dotted lines. However, it will come at a price: if an editable output file is produced (i.e., an output file in Illustrator, Postscript or Fig format), it will be difficulty to modify its dashed and dotted lines with a drawing editor.

'-f *font_size*'
'--font-size *font_size*'
: (Float, default 0.0175.) Set the size of the font used for rendering text, as a fraction of the width of the graphics display, to *font_size*.

'-F *font_name*'
'--font-name *font_name*'
: (String, default "Helvetica" except for pic2plot -T pnm, pic2plot -T gif, pic2plot -T pcl, pic2plot -T hpgl, pic2plot -T tek, and raw pic2plot, for all of which "HersheySerif" is the default.) Set the font used for text to *font_name*. Font names are case-insensitive. If the specified font is not available, the default font will be used. Which fonts are available depends on which '-T' option is used. For a list of all fonts, see Section A.1 [Text Fonts], page 124.

The plotfont utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

'-n'
'--no-centering'
Turn off the automatic centering of each figure. If this option is specified, the position of the objects in each figure may be specified in terms of absolute coordinates. E.g., "line from (0,0) to (4,4)" will draw a line segment from the lower left corner to the center of the graphics display, since the display width and display height are defined to equal 8 virtual inches.

'-W *line_width*'
'--line-width *line_width*'
(Float, default −1.0.) Set the default thickness of lines, as a fraction of the size (i.e., minimum dimension) of the graphics display, to *line_width*. A negative value means that the default value provided by the libplot graphics library should be used. This is usually 1/850 times the size of the display, although if '-T X', '-T pnm', or -T gif is specified, it is zero. By convention, a zero-thickness line is the thinnest line that can be drawn. This is the case in all output formats. Note, however, that the drawing editors idraw and xfig treat zero-thickness lines as invisible.

pic2plot -T hpgl does not support drawing lines with other than a default thickness if the environment variable HPGL_VERSION is set to a value less than "2" (the default).

'--bg-color *name*'
(String, default "white".) Set the color used for the background to be *name*. This is relevant only to pic2plot -T X, pic2plot -T pnm, and pic2plot -T gif. An unrecognized name sets the color to the default. For information on what names are recognized, see Appendix B [Color Names], page 146. The environment variable BG_COLOR can equally well be used to specify the background color.

If the '-T gif' option is used, a transparent pseudo-GIF may be produced by setting the TRANSPARENT_COLOR environment variable to the name of the background color. See Section 4.3 [pic2plot Environment], page 40.

'--bitmap-size *bitmap_size*'
(String, default "570x570".) Set the size of the graphics display in which the plot will be drawn, in terms of pixels, to be *bitmap_size*. This is relevant only to pic2plot -T X, pic2plot -T pnm, and pic2plot -T gif, for which the graphics display size can be expressed in terms of pixels. The environment variable BITMAPSIZE can equally well be used to specify the size.

The graphics display used by pic2plot -T X is an X window; i.e., one window for each figure. If you choose a rectangular (non-square) window size, the fonts in each figure will be scaled anisotropically, i.e., by different factors in the horizontal and vertical direction. This requires an X11R6 display. Any font that cannot be scaled in this way will be replaced by a default scalable font, such as the Hershey vector font "HersheySerif".

For backward compatibility, the X resource `Xplot.geometry`, which can be set by the user, may be used to set the window size, instead of '`--bitmap-size`' or `BITMAPSIZE`.

'`--max-line-length` *max_line_length*'
: (Integer, default 500.) Set the maximum number of points that a polygonal line may contain, before it is flushed to the display device, to equal *max_line_length*. If this flushing occurs, the polygonal line will be split into two or more sub-lines, though the splitting should not be noticeable.

The reason for splitting long polygonal lines is that some display devices (e.g., old Postscript printers and HP-GL pen plotters) have limited buffer sizes. The environment variable `MAX_LINE_LENGTH` can also be used to specify the maximum line length. This option has no effect on raw `pic2plot`, since it draws polylines in real time and has no buffer limitations.

'`--page-size` *pagesize*'
: (String, default "letter".) Set the size of the page on which each figure will be positioned. This is relevant only to `pic2plot -T ai`, `pic2plot -T ps`, `pic2plot -T fig`, `pic2plot -T pcl`, and `pic2plot -T hpgl`. "letter" means an 8.5 in by 11 in page. Any ISO page size in the range "a0"..."a4" or ANSI page size in the range "a"..."e" may be specified ("letter" is an alias for "a" and "tabloid" is an alias for "b"). "legal", "ledger", and "b5" are recognized page sizes also. The environment variable `PAGESIZE` can equally well be used to specify the page size.

For `pic2plot -T ai` and `pic2plot -T ps`, the graphics display within which each figure is drawn will be a square region centered on the specified page, occupying its full width (with allowance being made for margins). For `pic2plot -T fig`, it will be a square region of the same size, located in the upper left corner of an `xfig` display. For `pic2plot -T pcl` and `pic2plot -T hpgl`, the graphics display will be a square region of the same size, but may be positioned differently. Fine control its positioning on the page can be accomplished by setting certain environment variables (see Section 4.3 [pic2plot Environment], page 40).

'`--pen-color` *name*'
: (String, default "black".) Set the pen color to be *name*. An unrecognized name sets the pen color to the default. For information on what color names are recognized, see Appendix B [Color Names], page 146.

'`--rotation` *angle*'
: (Integer, default 0.) Set the rotation angle of each figure within the graphics display to be *angle* degrees. Recognized values are 0, 90, 180, and 270. The rotation is counterclockwise.

This option is used for switching between portrait mode and landscape mode. Postmodernists may also find it useful.

The following option is relevant only to raw `pic2plot`, i.e., relevant only if no display type or output format is specified with the '`-T`' option. In this case `pic2plot` outputs a graphics metafile, which may be translated to other formats by invoking `plot`.

'-O'
'--portable-output'
: Output the portable (human-readable) version of GNU metafile format, rather than a binary version (the default). This can also be requested by setting the environment variable `META_PORTABLE` to "yes".

The following options request information.

'--help'
: Print a list of command-line options, and then exit.

'--help-fonts'
: Print a table of available fonts, and then exit. The table will depend on which display type or output format is specified with the '-T' option. `pic2plot -T X`, `pic2plot -T ai`, `pic2plot -T ps`, and `pic2plot -T fig` each support the 35 standard Postscript fonts. `pic2plot -T ai`, `pic2plot -T pcl`, and `pic2plot -T hpgl` support the 45 standard PCL 5 fonts, and `pic2plot -T pcl` and `pic2plot -T hpgl` support a number of Hewlett–Packard vector fonts. All of the preceding, together with `pic2plot -T pnm`, `pic2plot -T gif`, and `pic2plot -T tek`, support a set of 22 Hershey vector fonts. Raw `pic2plot` in principle supports any of these fonts, since its output must be translated to other formats with `plot`. The `plotfont` utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

'--list-fonts'
: Like '--help-fonts', but lists the fonts in a single column to facilitate piping to other programs. If no display type or output format is specified with the '-T' option, the full set of supported fonts is listed.

'--version'
: Print the version number of `pic2plot` and the plotting utilities package, and exit.

4.3 Environment variables

The behavior of `pic2plot` is affected by several environment variables. We have already mentioned the environment variables `BITMAPSIZE`, `PAGESIZE`, `BG_COLOR`, and `MAX_LINE_LENGTH`. They serve as backups for the options '--bitmap-size', '--page-size', '--bg-color', and '--max-line-length'. The remaining environment variables are specific to individual output formats.

`pic2plot -T X`, which pops up a window on an X Window System display for each figure, checks the `DISPLAY` environment variable. The value of this variable determines the display on which the windows will be popped up.

`pic2plot -T pnm`, which produces output in Portable Anymap (PBM/PGM/PPM) format, is affected by the `PNM_PORTABLE` environment variable. If its value is "yes", the output file will be in the portable (human readable) version of PBM, PGM, or PPM format, rather than the default (binary) version.

`pic2plot -T gif`, which produces output in pseudo-GIF format, is affected by two environment variables. If the value of the `INTERLACE` variable is "yes", the pseudo-GIF output file will be in interlaced format. Also, if the value of the `TRANSPARENT_COLOR` environment variable is the name of a color that appears in the output file, that color will be treated as transparent by most applications that read GIF files. For information on what color names are recognized, see Appendix B [Color Names], page 146.

`pic2plot -T pcl`, which produces PCL 5 output for Hewlett–Packard printers and plotters, is affected by several environment variables. The position of the graphics display on the page can be adjusted by setting the `PCL_XOFFSET` and `PCL_YOFFSET` environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the `PCL_ROTATE` environment variable to "yes". Besides "no" and "yes", recognized values for this variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively.

The variable `PCL_ASSIGN_COLORS` is also recognized. It should be set to "yes" when producing PCL 5 output for a color printer or other color device. This will ensure accurate color reproduction by giving the output device complete freedom in assigning colors, internally, to its "logical pens". If it is "no" then the device will use a fixed set of colored pens, and will emulate other colors by shading. The default is "no" because monochrome PCL 5 devices, which are much more common than colored ones, must use shading to emulate color.

`pic2plot -T hpgl`, which produces Hewlett–Packard Graphics Language output, is also affected by several environment variables. The most important is `HPGL_VERSION`, which may be set to "1", "1.5", or "2" (the default). "1" means that the output should be generic HP-GL, "1.5" means that the output should be suitable for the HP7550A graphics plotter and the HP758x, HP7595A and HP7596A drafting plotters (HP-GL with some HP-GL/2 extensions), and "2" means that the output should be modern HP-GL/2. If the version is "1" or "1.5" then the only available fonts will be vector fonts, and all lines will be drawn with a default thickness (the '`-W`' option will not work). Additionally, if the version is "1" then the filling of arbitrary curves with solid color will not be supported (circles and rectangles aligned with the coordinate axes may be filled, though).

The position of the `pic2plot -T hpgl` graphics display on the page can be adjusted by setting the `HPGL_XOFFSET` and `HPGL_YOFFSET` environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the `HPGL_ROTATE` environment variable to "yes". Besides "no" and "yes", recognized values for this variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively. "180" and "270" are supported only if `HPGL_VERSION` is "2" (the default).

Opaque filling and the drawing of visible white lines are supported only if `HPGL_VERSION` is "2" (the default) and the environment variable `HPGL_OPAQUE_MODE` is "yes" (the default). If the value is "no" then opaque filling will not be used, and white lines (if any), which are normally drawn with pen #0, will not be drawn. This feature is to accommodate older HP-GL/2 devices. HP-GL/2 pen plotters, for example, do not support opacity or the use of pen #0 to draw visible white lines. Some older HP-GL/2 devices reportedly malfunction if asked to draw opaque objects.

By default, `pic2plot -T hpgl` will draw with a fixed set of pens. Which pens are present may be specified by setting the `HPGL_PENS` environment variable. If `HPGL_VERSION` is "1", the default value of `HPGL_PENS` is "1=black"; if `HPGL_VERSION` is "1.5" or "2", the default value of `HPGL_PENS` is "1=black:2=red:3=green:4=yellow:5=blue:6=magenta:7=cyan". The format should be self-explanatory. By setting `HPGL_PENS`, you may specify a color for any pen in the range #1...#31. For information on what color names are recognized, see Appendix B [Color Names], page 146. Pen #1 must always be present, though it need not be black. Any other pen in the range #1...#31 may be omitted.

If `HPGL_VERSION` is "2" then `pic2plot -T hpgl` will also be affected by the environment variable `HPGL_ASSIGN_COLORS`. If the value of this variable is "yes", then `plot -T hpgl` will not be restricted to the palette specified in `HPGL_PENS`: it will assign colors to "logical pens" in the range #1...#31, as needed. The default value is "no" because other than color LaserJet printers and DesignJet plotters, not many HP-GL/2 devices allow the assignment of colors to logical pens.

`pic2plot -T tek`, which produces output for a Tektronix terminal or emulator, checks the `TERM` environment variable. If the value of `TERM` is "xterm", "xterms", or "kterm", it is taken as a sign that the current application is running in an X Window System VT100 terminal emulator: an `xterm`. Before drawing graphics, `pic2plot -T tek` will emit an escape sequence that causes the terminal emulator's auxiliary Tektronix window, which is normally hidden, to pop up. After the graphics are drawn, an escape sequence that returns control to the original VT100 window will be emitted. The Tektronix window will remain on the screen.

If the value of `TERM` is "kermit", "ansi.sys", "ansissys", "ansi.sysk", or "ansisysk", it is taken as a sign that the current application is running in the VT100 terminal emulator provided by the MS-DOS version of `kermit`. Before drawing graphics, `pic2plot -T tek` will emit an escape sequence that switches the terminal emulator to Tektronix mode. Also, some of the Tektronix control codes emitted by `pic2plot -T tek` will be `kermit`-specific. There will be a limited amount of color support, which is not normally the case (the 16 `ansi.sys` colors will be supported). After drawing graphics, `pic2plot -T tek` will emit an escape sequence that returns the emulator to VT100 mode. The key sequence 'ALT minus' can be employed manually within `kermit` to switch between the two modes.

5 The tek2plot Program

5.1 What tek2plot is used for

GNU `tek2plot` is a command-line Tektronix translator. It displays Tektronix graphics files, or translates them to other formats. The '`-T`' option is used to specify the output format or display type. Supported output formats include "X", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", and "meta" (the default). These are the same formats that are supported by the GNU `graph`, `plot`, and `pic2plot` programs. `tek2plot` will take input from a file specified on the command line or from standard input, just as the plot filter `plot` does.

Tektronix graphics files are produced by many older applications, such as http://tdc-www.harvard.edu/software/skymap, SKYMAP, a powerful astronomical display program. A directory containing sample Tektronix graphics files, which you may experiment with, is distributed along with the GNU plotting utilities. On most systems it is installed as '`/usr/share/tek2plot`' or '`/usr/local/share/tek2plot`'.

Tektronix graphics format is defined as a noninteractive version of the graphics format understood by Tektronix 4010/4014 terminals, as documented in the *4014 Service Manual*, Tektronix Inc., 1974 (Tektronix Part #070-1648-00). `tek2plot` does not support interactive features such as graphics input mode ("GIN mode") or status enquiry. However, it does support a few additional features provided by popular Tektronix emulators, such as the color extensions supported by the Tektronix emulator contained in the MS-DOS version of `kermit`.

5.2 tek2plot command-line options

The `tek2plot` program translates the Tektronix graphics files produced by many older applications to other formats. The output format or display type is specified with the '`-T`' option. The possible output formats are the same ten formats that are supported by the GNU `graph`, `plot`, and `pic2plot` programs.

Input file names may be specified anywhere on the command line. That is, the relative order of file names and command-line options does not matter. If no file names are specified, or the file name '`-`' is specified, the standard input is read. The output file is written to standard output, unless the '`-T X`' option is specified. In that case the output is displayed in one or more windows on an X Window System display, and there is no output file.

The full set of command-line options is listed below. There are three sorts of option:

1. General options.
2. Options relevant only to raw `tek2plot`, i.e., relevant only if no display type or output format is specified with the '`-T`' option.
3. Options requesting information (e.g., '`--help`').

Each option that takes an argument is followed, in parentheses, by the type and default value of the argument.

The following are general options.

'-T *type*'
'--display-type *type*'

(String, default "meta".) Select a display type or output format of type *type*, which may be one of the strings "X", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", and "meta". These refer respectively to the X Window System, portable anymap (PBM/PGM/PPM) format, pseudo-GIF format, the format used by Adobe Illustrator, `idraw`-editable Postscript, the format used by the `xfig` drawing editor, the Hewlett–Packard PCL 5 printer language, the Hewlett–Packard Graphics Language (by default, HP-GL/2), Tektronix format, and device-independent GNU graphics metafile format.

'-p *n*'
'--page-number *n*'

(Nonnegative integer.) Display only page number *n*, within the Tektronix file or sequence of Tektronix files that is being translated. Tektronix files may consist of one or more pages, numbered beginning with zero.

The default behavior, if the '-p' option is not used, is to display all nonempty pages in succession. For example, `tek2plot -T X` displays each page in its own X window. If the '-T pnm' option, the '-T gif' option, the '-T ai' option, or the '-T fig' option is used, the default behavior is to display only the first page, since files in PNM, pseudo-GIF, AI, or Fig format may contain only a single page of graphics.

Most Tektronix files consist of either one page (page #0) or two pages (an empty page #0, and page #1). Tektronix files produced by the GNU plotting utilities (e.g., by `graph -T tek`) are normally of the latter sort.

'-F *font_name*'
'--font-name *font_name*'

(String, default "Courier" except for `tek2plot -T pnm`, `tek2plot -T gif`, `tek2plot -T pcl`, `tek2plot -T hpgl`, and raw `tek2plot`, for all of which "HersheySerif" is the default.) Set the font used for text to *font_name*. Font names are case-insensitive. If a font outside the Courier family is chosen, the '--position-chars' option (see below) should probably be used. For a list of all fonts, see Section A.1 [Text Fonts], page 124. If the specified font is not available, the default font will be used.

'-W *line_width*'
'--line-width *line_width*'

(Float, default −1.0.) Set the thickness of lines, as a fraction of the size (i.e., minimum dimension) of the graphics display, to *line_width*. A negative value means that the default value provided by the `libplot` graphics library should be used. This is usually 1/850 times the size of the display, although if '-T X', '-T pnm', or `-T gif` is specified, it is zero. By convention, a zero-thickness line is the thinnest line that can be drawn. This is the case in all output formats. Note, however, that the drawing editors `idraw` and `xfig` treat zero-thickness lines as invisible.

tek2plot -T hpgl does not support drawing lines with other than a default thickness if the environment variable HPGL_VERSION is set to a value less than "2" (the default).

'--bg-color *name*'
: (String, default "white".) Set the color used for the background to be *name*. This is relevant only to tek2plot -T X, tek2plot -T pnm, and tek2plot -T gif. An unrecognized name sets the color to the default. For information on what names are recognized, see Appendix B [Color Names], page 146. The environment variable BG_COLOR can equally well be used to specify the background color.

  If the '-T gif' option is used, a transparent pseudo-GIF may be produced by setting the TRANSPARENT_COLOR environment variable to the name of the background color. See Section 5.3 [tek2plot Environment], page 47.

'--bitmap-size *bitmap_size*'
: (String, default "570x570".) Set the size of the graphics display in which the plot will be drawn, in terms of pixels, to be *bitmap_size*. This is relevant only to tek2plot -T X, tek2plot -T pnm, and tek2plot -T gif, for which the graphics display size can be expressed in terms of pixels. The environment variable BITMAPSIZE can equally well be used to specify the size.

  The graphics display used by tek2plot -T X is an X window. If you choose a rectangular (non-square) window size, the fonts in the plot will be scaled anisotropically, i.e., by different factors in the horizontal and vertical direction. This requires an X11R6 display. Any font that cannot be scaled in this way will be replaced by a default scalable font, such as the Hershey vector font "HersheySerif".

  For backward compatibility, the X resource Xplot.geometry, which can be set by the user, may be used to set the window size, instead of '--bitmap-size' or BITMAPSIZE.

'--max-line-length *max_line_length*'
: (Integer, default 500.) Set the maximum number of points that a polygonal line may contain, before it is flushed to the display device, to equal *max_line_length*. If this flushing occurs, the polygonal line will be split into two or more sub-lines, though the splitting should not be noticeable.

  The reason for splitting long polygonal lines is that some display devices (e.g., old Postscript printers and HP-GL pen plotters) have limited buffer sizes. The environment variable MAX_LINE_LENGTH can also be used to specify the maximum line length. This option has no effect on raw tek2plot, since it draws polylines in real time and has no buffer limitations.

'--page-size *pagesize*'
: (String, default "letter".) Set the size of the page on which the plot will be positioned. This is relevant only to tek2plot -T ai, tek2plot -T ps, tek2plot -T fig, tek2plot -T pcl, and tek2plot -T hpgl. "letter" means an 8.5 in by 11 in page. Any ISO page size in the range "a0"..."a4" or ANSI page size in the range "a"..."e" may be specified ("letter" is an alias for "a" and "tabloid"

is an alias for "b"). "legal", "ledger", and "b5" are recognized page sizes also. The environment variable `PAGESIZE` can equally well be used to specify the page size.

For `tek2plot -T ai` and `tek2plot -T ps`, the graphics display within which the plot is drawn will be a square region centered on the specified page, occupying its full width (with allowance being made for margins). For `tek2plot -T fig`, it will be a square region of the same size, located in the upper left corner of an `xfig` display. For `tek2plot -T pcl` and `tek2plot -T hpgl`, the graphics display will be a square region of the same size, but may be positioned differently. Fine control its positioning on the page can be accomplished by setting certain environment variables (see Section 5.3 [tek2plot Environment], page 47).

'`--pen-color` *name*'

(String, default "black".) Set the pen color to be *name*. An unrecognized name sets the pen color to the default. For information on what color names are recognized, see Appendix B [Color Names], page 146.

'`--position-chars`'

Position the characters in each text string individually on the display. If the text font is not a member of the Courier family, and especially if it is not a fixed-width font, this option is recommended. It will improve the appearance of text strings, at the price of making it difficult to edit the output file with `xfig` or `idraw`.

'`--rotation` *angle*'

(Integer, default 0.) Set the rotation angle of the plot within the graphics display to be *angle* degrees. Recognized values are 0, 90, 180, and 270. The rotation is counterclockwise.

This option is used for switching between portrait mode and landscape mode. Postmodernists may also find it useful.

'`--use-tek-fonts`'

Use the bitmap fonts that were used on the original Tektronix 4010/4014 terminal. This option is relevant only to `tek2plot -T X`. The four relevant bitmap fonts are distributed with most versions of the plotting utilities package, under the names `tekfont0`...`tekfont3`. They may easily be installed on any modern X Window System display. For this option to work properly, you must also select a window size of 1024x1024 pixels, either by using the `--bitmap-size 1024x1024` option or by setting the value of the `Xplot.geometry` resource. The reason for this restriction is that bitmap fonts, unlike the scalable fonts that the plotting utilities normally use, cannot be rescaled.

This option is useful only if you have a file in Tektronix format that draws text using native Tektronix fonts. Tektronix files produced by the GNU plotting utilities (e.g., by `graph -T tek`) do not use native Tektronix fonts to draw text.

The following option is relevant only to raw `tek2plot`, i.e., relevant only if no display type or output format is specified with the '`-T`' option. In this case `tek2plot` outputs a graphics metafile, which may be translated to other formats by invoking `plot`.

'-O'
'--portable-output'
: Output the portable (human-readable) version of GNU metafile format, rather than a binary version (the default). This can also be requested by setting the environment variable META_PORTABLE to "yes".

The following options request information.

'--help' Print a list of command-line options, and then exit.

'--help-fonts'
: Print a table of available fonts, and then exit. The table will depend on which display type or output format is specified with the '-T' option. tek2plot -T X, tek2plot -T ai, tek2plot -T ps, and tek2plot -T fig each support the 35 standard Postscript fonts. tek2plot -T ai, tek2plot -T pcl, and tek2plot -T hpgl support the 45 standard PCL 5 fonts, and tek2plot -T pcl and tek2plot -T hpgl support a number of Hewlett–Packard vector fonts. All of the preceding, together with tek2plot -T pnm, tek2plot -T gif, and tek2plot -T tek, support a set of 22 Hershey vector fonts. Raw tek2plot in principle supports any of these fonts, since its output must be translated to other formats with plot. The plotfont utility will produce a character map of any available font. See Chapter 6 [plotfont], page 50.

'--list-fonts'
: Like '--help-fonts', but lists the fonts in a single column to facilitate piping to other programs. If no display type or output format is specified with the '-T' option, the full set of supported fonts is listed.

'--version'
: Print the version number of tek2plot and the plotting utilities package, and exit.

5.3 Environment variables

The behavior of tek2plot is affected by several environment variables, which are the same as those that affect graph and plot. For convenience, we list them here.

We have already mentioned the environment variables BITMAPSIZE, PAGESIZE, BG_COLOR, and MAX_LINE_LENGTH. They serve as backups for the options '--bitmap-size', '--page-size', '--bg-color', and '--max-line-length'. The remaining environment variables are specific to individual output formats.

tek2plot -T X, which pops up a window on an X Window System display and draws graphics in it, checks the DISPLAY environment variable. The value of this variable determines the display on which the window will be popped up.

tek2plot -T pnm, which produces output in Portable Anymap (PBM/PGM/PPM) format, is affected by the PNM_PORTABLE environment variable. If its value is "yes", the output file will be in the portable (human readable) version of PBM, PGM, or PPM format, rather than the default (binary) version.

tek2plot -T gif, which produces output in pseudo-GIF format, is affected by two environment variables. If the value of the INTERLACE variable is "yes", the pseudo-GIF output file will be in interlaced format. Also, if the value of the TRANSPARENT_COLOR environment variable is the name of a color that appears in the output file, that color will be treated as transparent by most applications that read GIF files. For information on what color names are recognized, see Appendix B [Color Names], page 146.

tek2plot -T pcl, which produces PCL 5 output for Hewlett–Packard printers and plotters, is affected by several environment variables. The position of the graphics display on the page can be adjusted by setting the PCL_XOFFSET and PCL_YOFFSET environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the PCL_ROTATE environment variable to "yes". This is not the same as the rotation obtained with the --rotation option, which sets the rotation angle of the plot within the display. Besides "no" and "yes", recognized values for the PCL_ROTATE variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively.

The variable PCL_ASSIGN_COLORS is also recognized. It should be set to "yes" when producing PCL 5 output for a color printer or other color device. This will ensure accurate color reproduction by giving the output device complete freedom in assigning colors, internally, to its "logical pens". If it is "no" then the device will use a fixed set of colored pens, and will emulate other colors by shading. The default is "no" because monochrome PCL 5 devices, which are much more common than colored ones, must use shading to emulate color.

tek2plot -T hpgl, which produces Hewlett–Packard Graphics Language output, is also affected by several environment variables. The most important is HPGL_VERSION, which may be set to "1", "1.5", or "2" (the default). "1" means that the output should be generic HP-GL, "1.5" means that the output should be suitable for the HP7550A graphics plotter and the HP758x, HP7595A and HP7596A drafting plotters (HP-GL with some HP-GL/2 extensions), and "2" means that the output should be modern HP-GL/2. If the version is "1" or "1.5" then the only available fonts will be vector fonts, and all lines will be drawn with a default thickness (the '-W' option will not work).

The position of the tek2plot -T hpgl graphics display on the page can be adjusted by setting the HPGL_XOFFSET and HPGL_YOFFSET environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the HPGL_ROTATE environment variable to "yes". This is not the same as the rotation obtained with the --rotation option, which sets the rotation angle of the plot within the display. Besides "no" and "yes", recognized values for the HPGL_ROTATE variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively. "180" and "270" are supported only if HPGL_VERSION is "2" (the default).

The drawing of visible white lines is supported only if HPGL_VERSION is "2" and the environment variable HPGL_OPAQUE_MODE is "yes" (the default). If the value is "no" then white lines (if any), which are normally drawn with pen #0, will not be drawn. This feature is to accommodate older HP-GL/2 devices. HP-GL/2 pen plotters, for example, do not support the use of pen #0 to draw visible white lines. Some older HP-GL/2 devices may, in fact, malfunction if asked to draw opaque objects.

By default, tek2plot -T hpgl will draw with a fixed set of pens. Which pens are present may be specified by setting the HPGL_PENS environment variable. If HPGL_VERSION is "1", the default value of HPGL_PENS is "1=black"; if HPGL_VERSION is "1.5" or "2", the default value of HPGL_PENS is "1=black:2=red:3=green:4=yellow:5=blue:6=magenta:7=cyan". The format should be self-explanatory. By setting HPGL_PENS, you may specify a color for any pen in the range #1...#31. For information on what color names are recognized, see Appendix B [Color Names], page 146. Pen #1 must always be present, though it need not be black. Any other pen in the range #1...#31 may be omitted.

If HPGL_VERSION is "2" then tek2plot -T hpgl will also be affected by the environment variable HPGL_ASSIGN_COLORS. If the value of this variable is "yes", then tek2plot -T hpgl will not be restricted to the palette specified in HPGL_PENS: it will assign colors to “logical pens” in the range #1...#31, as needed. The default value is "no" because other than color LaserJet printers and DesignJet plotters, not many HP-GL/2 devices allow the assignment of colors to logical pens.

6 The plotfont Utility

6.1 How to use plotfont

GNU `plotfont` is a simple utility that will produce a character map for any font available to the GNU plotting utilities `graph`, `plot`, `pic2plot`, and `tek2plot`, and the GNU `libplot` graphics library on which they are based. The map may be displayed on an X Window System display, or produced in any of several output formats. The '`-T`' option is used to specify the desired output format. Supported output formats include "X", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", and "meta" (the default).

Which fonts are available depends on the choice of display or output format. To get a list of the available fonts, use the '`--help-fonts`' option. For example,

```
plotfont -T ps --help-fonts
```

will list the fonts that are available when producing Postscript output. One of these fonts is "Times-Roman". Doing

```
plotfont -T ps Times-Roman > map.ps
```

will produce a character map of the lower half of this font, which consists of printable ASCII characters. The map will be a 12x8 grid, with a character centered in each grid cell. If you include the '`-2`' option, you will get a map of the upper half of the font.

Most built-in fonts are ISO-Latin-1 fonts, which means that the upper half is arranged according to the ISO-Latin-1 encoding. The "HersheyCyrillic" font is one that is not. If you do

```
plotfont -T ps -2 HersheyCyrillic > map.ps
```

you will get a map that illustrates its arrangment, which is called KOI8-R. The KOI8-R arrangement is the standard for Unix and networking applications in the former Soviet Union. So-called dingbats fonts, such as "ZapfDingbats" and "Wingdings", also have an individualistic layout. In most installations of the plotting utilities, the Wingdings font is not available when producing Postscript output. However, it is available when producing output in PCL 5 or HP-GL/2 format. If you do

```
plotfont -T hpgl Wingdings > map.plt
```

you will get a Wingdings character map, in HP-GL/2 format, that may be imported into any application that understands HP-GL/2. Similarly, `plot -T pcl Wingdings` will produce a Wingdings character map in PCL 5 format, which may be printed on a LaserJet or other PCL 5 device.

In all, more than a hundred fonts are built into the plotting utilities. See Section A.1 [Text Fonts], page 124. Actually, if you are using the plotting utilities to display output on an X display, you are not restricted to the built-in fonts. Doing

```
plotfont -T X --help-fonts
```

produces a list of the built-in fonts that are available, including both Hershey and Postscript fonts. But fonts available on your X display may also be used. The `xlsfonts` command will list the fonts available on your X display, most font names being given in what is called XLFD format. The plotting utilities refer to X fonts by shortened versions of their XLFD names. For example, the font "Utopia-Regular" is available on many X displays. Its XLFD name is "-adobe-utopia-medium-r-normal–0-0-0-0-p-0-iso8859-1", and its shortened XLFD name is "utopia-medium-r-normal". If you do

```
plotfont -T X utopia-medium-r-normal
```

then a character map for this font will be displayed in a popped-up X window.

When using the '-T X' option, you may also use the '--bitmap-size' option to choose the size of the popped-up window. Modern X displays can scale fonts by different amounts in the horizontal and vertical directions. If, for example, you add '--bitmap-size 600x300' to the above command line, both the character map and the Utopia-Regular font within it will be scaled in this way. If your X display does not support font scaling, a scalable font will be substituted.

6.2 plotfont command-line options

The plotfont font display utility will produce a character map for any of the fonts available to the GNU plotting utilities graph, plot, pic2plot, and tek2plot, and the GNU libplot graphics library on which they are based. The map may be produced in any supported output format, or displayed on an X Window System display. The output format or display type is specified with the '-T' option.

The names of the fonts for which a character map will be produced may appear anywhere on the plotfont command line. That is, the relative order of font names and command-line options does not matter. The character map is written to standard output, unless the '-T X' option is specified. In that case the character map is displayed in a window on an X Window System display, and there is no output file.

The possible options are listed below. There are three sorts of option:

1. General options.
2. Options relevant only to raw plotfont, i.e., relevant only if no display type or output format is specified with the '-T' option.
3. Options requesting information (e.g., '--help').

Each option that takes an argument is followed, in parentheses, by the type and default value of the argument.

The following are general options.

'-1'
'--lower-half'
: Generate a character map for the lower half of each specified font. This is the default.

'-2'
'--upper-half'
: Generate a character map for the upper half of each specified font.

'-o'
'--octal'
: Number the characters in octal rather than in decimal (the default).

'-x'
'--hexadecimal'
: Number the characters in hexadecimal rather than in decimal (the default).

'`--box`' Surround each character with a box, showing its extent to left and right. The default is not to do this.

'`-j` *row*'

'`--jis-row` *row*'

Generate a character map for row *row* of a Japanese font arranged according to JIS [Japanese Industrial Standard] X0208. The only such font currently available is the HersheyEUC [Extended Unix Code] font. If used, this option overrides the '`-1`' and '`-2`' options.

The valid rows are 1...94. In the JIS X0208 standard, Roman characters are located in row 3, and Japanese syllabic characters (Hiragana and Katakana) are located in rows 4 and 5. Greek and Cyrillic characters are located in rows 6 and 7. Japanese ideographic characters (Kanji) are located in rows 16...84. Rows 16...47 contain the JIS Level 1 Kanji, which are the most frequently used. They are arranged according to On (old Chinese) reading. Rows 48...84 contain the less frequently used JIS Level 2 Kanji.

The HersheyEUC font contains 596 of the 2965 Level 1 Kanji, and seven of the Level 2 Kanji. It uses the 8-bit EUC-JP encoding. This encoding is a multibyte encoding that includes the ASCII character set as well as the JIS X0208 characters. It represents each ASCII character in the usual way, i.e., as a single byte that does not have its high bit set. Each JIS X0208 character is represented as two bytes, each with the high bit set. The first byte contains the row number (plus 32), and the second byte contains the character number.

'`-T` *type*'

'`--display-type` *type*'

(String, default "meta".) Select a display type or output format of type *type*, which may be one of the strings "X", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", and "meta". These refer respectively to the X Window System, portable anymap (PBM/PGM/PPM) format, pseudo-GIF format, the format used by Adobe Illustrator, `idraw`-editable Postscript, the format used by the `xfig` drawing editor, the Hewlett–Packard PCL 5 printer language, the Hewlett–Packard Graphics Language (by default, HP-GL/2), Tektronix format, and device-independent GNU graphics metafile format.

Files in PNM, pseudo-GIF, AI, or Fig format may contain only a single page of graphics. So if the '`-T pnm`' option, the '`-T gif`' option, the '`-T ai`' option, or the '`-T fig`' option is used, a character map will be produced for only the first-specified font.

'`--bg-color` *name*'

(String, default "white".) Set the color used for the background to be *name*. This is relevant only to `plotfont -T X`, `plotfont -T pnm`, and `plotfont -T gif`. An unrecognized name sets the color to the default. For information on what names are recognized, see Appendix B [Color Names], page 146. The environment variable `BG_COLOR` can equally well be used to specify the background color.

If the '-T gif' option is used, a transparent pseudo-GIF may be produced by setting the TRANSPARENT_COLOR environment variable to the name of the background color. See Section 6.3 [plotfont Environment], page 55.

'--bitmap-size *bitmap_size*'

(String, default "570x570".) Set the size of the graphics display in which the plot will be drawn, in terms of pixels, to be *bitmap_size*. This is relevant only to plotfont -T X, plotfont -T pnm, and plotfont -T gif, for which the graphics display size can be expressed in terms of pixels. The environment variable BITMAPSIZE can equally well be used to specify the size.

The graphics display used by plotfont -T X is an X window. If you choose a rectangular (non-square) window size, the fonts in the plot will be scaled anisotropically, i.e., by different factors in the horizontal and vertical direction. This requires an X11R6 display. Any font that cannot be scaled in this way will be replaced by a default scalable font, such as the Hershey vector font "HersheySerif".

For backward compatibility, the X resource Xplot.geometry, which can be set by the user, may be used to set the window size, instead of '--bitmap-size' or BITMAPSIZE.

'--numbering-font-name *font_name*'

(String, default "Helvetica" except for plotfont -T pnm, plotfont -T gif, plotfont -T pcl, plotfont -T hpgl and plotfont -T tek, for which "HersheySerif" is the default.) Set the font used for the numbering of the characters in the character map(s) to be *font_name*.

'--page-size *pagesize*'

(String, default "letter".) Set the size of the page for which the character map(s) will be produced. This is relevant only to plotfont -T ai, plotfont -T ps, plotfont -T fig, plotfont -T pcl and plotfont -T hpgl. "letter" means an 8.5 in by 11 in page. Any ISO page size in the range "a0"..."a4" or ANSI page size in the range "a"..."e" may be specified ("letter" is an alias for "a" and "tabloid" is an alias for "b"). "legal", "ledger", and "b5" are recognized page sizes also. The environment variable PAGESIZE can equally well be used to specify the page size.

For plotfont -T ai and plotfont -T ps, the graphics display within which each character map is drawn will be a square region centered on the specified page, occupying its full width (with allowance being made for margins). For plotfont -T fig, it will be a square region of the same size, located in the upper left corner of an xfig display. For plotfont -T pcl and plotfont -T hpgl, the graphics display will be a square region of the same size, but may be positioned differently. Fine control over its positioning on the page can be accomplished by setting certain environment variables (see Section 6.3 [plotfont Environment], page 55).

'`--pen-color` *name*'
: (String, default "black".) Set the pen color to be *name*. An unrecognized name sets the pen color to the default. For information on what color names are recognized, see Appendix B [Color Names], page 146.

'`--rotation` *angle*'
: (Integer, default 0.) Set the rotation angle of the character map(s) within the graphics display to be *angle* degrees. Recognized values are 0, 90, 180, and 270. The rotation is counterclockwise.

  This option is used for switching between portrait mode and landscape mode. Postmodernists may also find it useful.

'`--title-font-name` *font_name*'
: (String) Set the font used for the title of each character map to be *font_name*. Normally the font used for the title is the same as the font whose character set is being displayed. This option is useful when producing character maps for unusual fonts such as "ZapfDingbats" and "Wingdings".

The following option is relevant only to raw `plotfont`, i.e., relevant only if no display type or output format is specified with the '`-T`' option. In this case `plotfont` outputs a graphics metafile, which may be translated to other formats by invoking `plot`.

'`-O`'
'`--portable-output`'
: Output the portable (human-readable) version of GNU metafile format, rather than a binary version (the default). This can also be requested by setting the environment variable `META_PORTABLE` to "yes".

The following options request information.

'`--help`'
: Print a list of command-line options, and then exit.

'`--help-fonts`'
: Print a table of available fonts, and then exit. The table will depend on which display type or output format is specified with the '`-T`' option. `plotfont -T X`, `plotfont -T ai`, `plotfont -T ps`, and `plotfont -T fig` each support the 35 standard Postscript fonts. `plotfont -T ai`, `plotfont -T pcl`, and `plotfont -T hpgl` support the 45 standard PCL 5 fonts, and `plotfont -T pcl` and `plotfont -T hpgl` support a number of Hewlett–Packard vector fonts. All of the preceding, together with `plotfont -T pnm`, `plotfont -T gif`, and `plotfont -T tek`, support a set of 22 Hershey vector fonts. Raw `plotfont` in principle supports any of these fonts, since its output must be translated to other formats with `plot`.

'`--list-fonts`'
: Like '`--help-fonts`', but lists the fonts in a single column to facilitate piping to other programs. If no display type or output format is specified with the '`-T`' option, the full set of supported fonts is listed.

'`--version`'
: Print the version number of `plotfont` and the plotting utilities package, and exit.

6.3 Environment variables

The behavior of `plotfont` is affected by several environment variables, which are the same as those that affect `graph`, `plot`, and `tek2plot`. For convenience, we list them here.

We have already mentioned the environment variables `BITMAPSIZE`, `PAGESIZE`, and `BG_COLOR`. They serve as backups for the options '`--bitmap-size`', '`--page-size`', and '`--bg-color`'. The remaining environment variables are specific to individual output formats.

`plotfont -T X`, which pops up a window on an X Window System display and draws a character map in it, checks the `DISPLAY` environment variable. The value of this variable determines the display on which the window will be popped up.

`plotfont -T pnm`, which produces output in Portable Anymap (PBM/PGM/PPM) format, is affected by the `PNM_PORTABLE` environment variable. If its value is "yes", the output file will be in the portable (human readable) version of PBM, PGM, or PPM format, rather than the default (binary) version.

`plotfont -T gif`, which produces output in pseudo-GIF format, is affected by two environment variables. If the value of the `INTERLACE` variable is "yes", the pseudo-GIF output file will be in interlaced format. Also, if the value of the `TRANSPARENT_COLOR` environment variable is the name of a color that appears in the output file, that color will be treated as transparent by most applications that read GIF files. For information on what color names are recognized, see Appendix B [Color Names], page 146.

`plotfont -T pcl`, which produces PCL 5 output for Hewlett–Packard printers and plotters, is affected by several environment variables. The position of the graphics display on the page can be adjusted by setting the `PCL_XOFFSET` and `PCL_YOFFSET` environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the `PCL_ROTATE` environment variable to "yes". This is not the same as the rotation obtained with the `--rotation` option, which sets the rotation angle of the character map within the display. Besides "no" and "yes", recognized values for the `PCL_ROTATE` variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively.

The variable `PCL_ASSIGN_COLORS` is also recognized. It should be set to "yes" when producing PCL 5 output for a color printer or other color device. This will ensure accurate color reproduction by giving the output device complete freedom in assigning colors, internally, to its "logical pens". If it is "no" then the device will use a fixed set of colored pens, and will emulate other colors by shading. The default is "no" because monochrome PCL 5 devices, which are much more common than colored ones, must use shading to emulate color.

`plotfont -T hpgl`, which produces Hewlett–Packard Graphics Language output, is also affected by several environment variables. The most important is `HPGL_VERSION`, which may be set to "1", "1.5", or "2" (the default). "1" means that the output should be generic HP-GL, "1.5" means that the output should be suitable for the HP7550A graphics plotter and the HP758x, HP7595A and HP7596A drafting plotters (HP-GL with some HP-GL/2 extensions), and "2" means that the output should be modern HP-GL/2. If the version is "1" or "1.5" then the only available fonts will be vector fonts.

The position of the `plotfont -T hpgl` graphics display on the page can be adjusted by setting the `HPGL_XOFFSET` and `HPGL_YOFFSET` environment variables, which may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in". Also, the display can be rotated 90 degrees counterclockwise on the page by setting the `HPGL_ROTATE` environment variable to "yes". This is not the same as the rotation obtained with the `--rotation` option, which sets the rotation angle of the character map within the display. Besides "no" and "yes", recognized values for the `HPGL_ROTATE` variable are "0", "90", "180", and "270". "no" and "yes" are equivalent to "0" and "90", respectively. "180" and "270" are supported only if `HPGL_VERSION` is "2" (the default).

By default, `plotfont -T hpgl` will draw with a fixed set of pens. Which pens are present may be specified by setting the `HPGL_PENS` environment variable. If `HPGL_VERSION` is "1", the default value of `HPGL_PENS` is "1=black"; if `HPGL_VERSION` is "1.5" or "2", the default value of `HPGL_PENS` is "1=black:2=red:3=green:4=yellow:5=blue:6=magenta:7=cyan". The format should be self-explanatory. By setting `HPGL_PENS`, you may specify a color for any pen in the range #1...#31. For information on what color names are recognized, see Appendix B [Color Names], page 146. Pen #1 must always be present, though it need not be black. Any other pen in the range #1...#31 may be omitted.

If `HPGL_VERSION` is "2" then `plotfont -T hpgl` will also be affected by the environment variable `HPGL_ASSIGN_COLORS`. If the value of this variable is "yes", then `plotfont -T hpgl` will not be restricted to the palette specified in `HPGL_PENS`: it will assign colors to "logical pens" in the range #1...#31, as needed. The default value is "no" because other than color LaserJet printers and DesignJet plotters, not many HP-GL/2 devices allow the assignment of colors to logical pens.

`plotfont -T tek`, which produces output for a Tektronix terminal or emulator, checks the `TERM` environment variable. If the value of `TERM` is "xterm", "xterms", or "kterm", it is taken as a sign that the current application is running in an X Window System VT100 terminal emulator: an `xterm`. Before drawing graphics, `plotfont -T tek` will emit an escape sequence that causes the terminal emulator's auxiliary Tektronix window, which is normally hidden, to pop up. After the graphics are drawn, an escape sequence that returns control to the original VT100 window will be emitted. The Tektronix window will remain on the screen.

If the value of `TERM` is "kermit", "ansi.sys", "ansissys", "ansi.sysk", or "ansisysk", it is taken as a sign that the current application is running in the VT100 terminal emulator provided by the MS-DOS version of `kermit`. Before drawing graphics, `plotfont -T tek` will emit an escape sequence that switches the terminal emulator to Tektronix mode. Also, some of the Tektronix control codes emitted by `plotfont -T tek` will be `kermit`-specific. There will be a limited amount of color support, which is not normally the case (the 16 `ansi.sys` colors will be supported). After drawing graphics, `plotfont -T tek` will emit an escape sequence that returns the emulator to VT100 mode. The key sequence 'ALT minus' can be employed manually within `kermit` to switch between the two modes.

7 The spline Program

7.1 How to use spline

GNU `spline` is a program for interpolating between the data points in one or more datasets. Each dataset would consist of values for an independent variable and a dependent variable, which may be a vector of specified fixed length. When discussing interpolation, we call these variables 't' and 'y', respectively. To emphasize: t is a scalar, but in general the dependent variable y may be a vector.

The simplest case is when there is a single input file, which is in ASCII format, and the vector y is one-dimensional. This is the default. For example, the input file could contain the dataset

```
0.0  0.0
1.0  1.0
2.0  0.0
```

which are the coordinates (t, y) of the data points (0,0), (1,1), and (2,0). Data points do not need to be on different lines, nor do the t and y coordinates of a data point need to be on the same line. However, there should be no blank lines in the input if it is to be viewed as forming a single dataset. Also, by default the t coordinate should be monotonically increasing, so that y may be viewed as a function of t.

You would construct a spline (the graph of an 'interpolating function') passing through the points in this dataset by doing

```
spline input_file > output_file
```

To produce a Postscript plot of the spline with the `graph` utility, you would do

```
spline input_file | graph -T ps > output.ps
```

To display a spline on an X Window System display, you could do

```
echo 0 0 1 1 2 0 | spline | graph -T X
```

Notice that the last example avoids the use of the input file altogether. `spline` will read from standard input if no files are specified on the command line, or if the special file name '`-`' is specified.

What exactly does `spline` do? First, it fits a curve (the graph of an interpolating function) through the points in the dataset. Then it splits the interval over which the independent variable t ranges into 100 sub-intervals, and computes the y values at each of the 101 subdivision points. It then outputs each of the pairs (t, y). These are the coordinates of 101 points that lie along a curve that interpolates between the points in the dataset. If there is more than one dataset in the input (separated by blank lines), each dataset is interpolated separately.

You may use the '`-n`' option to replace '100' by any other positive integer. You may also use the '`-t`' option to specify an interpolation interval that differs from the default (the interval over which the independent variable ranges). For example, the command

```
echo 0 0 1 1 2 0 | spline -n 20 -t 1.0 1.5 > output_file
```

will produce a dataset consisting of 21 (rather than 101) data points, with t values spaced regularly between 1.0 and 1.5 (rather than between 0.0 and 2.0). The data points will lie along a curve passing through (0,0), (1,1), and (2,0). This curve will be a parabola.

In general, the interpolating function will be a piecewise cubic spline. That is, between each pair of adjacent 'knots' (points in the input dataset), y will be a cubic function of t. This function will differ, depending on which pair of knots y lies between. At each knot, both the slope and curvature of the cubic pieces to either side will match. In mathematical terms, the interpolating curve will be twice continuously differentiable.

`spline` supports 'adding tension' to the interpolating curve. A nonzero value for the tension can be specified with the '`-T`' option. For example, a spline under considerable tension can be computed and displayed by doing

```
echo 0 0 1 0 2 0 | spline -T 10 | graph -T X
```

As the tension parameter is increased to positive infinity, the spline will converge to a polygonal line. You are meant to think of the spline as being drawn taut. Actually, tension may be negative as well as positive. A spline with negative tension will tend to bow outward, in fact to oscillate sinusoidally. But as the tension decreases to negative infinity, the spline, though oscillatory, will again converge to a polygonal line.

If the tension is positive, its reciprocal will be the maximum range of the independent variable t over which the spline will 'like to curve'. Increasing the tension far above zero will accordingly force the spline to consist of short curved sections, centered on the data points, and sections that are almost straight. It follows that tension is a 'dimensionful' quantity. If the tension is nonzero, then when the values of the independent variable are multiplied by some common positive factor, the tension should be divided by the same factor to obtain a scaled version of the original spline. If the tension is zero (the default, or cubic spline case), then the computation of the spline will be unaffected by linear scaling of the data.

In mathematical terms, a spline under tension will satisfy the differential equation

$$y'''' = \mathrm{sgn}(\mathit{tension})\,\mathit{tension}^2\, y''$$

between each successive pair of knots. If the tension equals zero, which is the default, the fourth derivative of y with respect to t will equal zero at every point. In this case, y as a function of t will reduce to a cubic polynomial between each successive pair of knots. But if the tension is nonzero, y will not be a polynomial function of t. It may be expressed in terms of exponential functions, however.

Irrespective of whether or not the spline is under tension, you may specify the '`-p`' option if you wish the spline to be a periodic function of t. This will only work if the y values for the first and last points in the dataset are equal. Otherwise, it would make no sense to compute a periodic interpolation.

It is sometimes useful to interpolate between data points at the same time as they are generated by an auxiliary program. That is, it is useful for `spline` to function as a real-time filter. `spline` does not normally act as a filter, since computing an interpolating curve that is as smooth as possible is a global task. But if the '`-f`' option is specified, `spline` will indeed function as a filter. A different interpolation algorithm (cubic Bessel interpolation, which is local rather than global) will be used. If '`-f`' is specified, '`-p`' may not be specified.

Also, if '-f' is specified then an interpolation interval (a range of t values) must be requested explicitly with the '-t' option.

Cubic Bessel interpolation is inherently less smooth than the construction of a global cubic spline. If the '-f' option is specified, the slope of the spline at each knot will be chosen by fitting a parabola through that knot, and the two adjacent knots. The slopes of the two interpolating segments to either side of each interior knot will match at that knot, but typically their curvatures will not. In mathematical terms, the interpolating curve will be continuously differentiable, but in general not twice continuously differentiable. This loss of differentiability is the price that is paid for functioning as a real-time filter.

7.2 Advanced use of `spline`

The preceding section explains how `spline` can be employed to interpolate a function y of a scalar variable t, in the case when y is a scalar. In this section we explain how to perform more sophisticated interpolations. This includes multidimensional interpolations, and interpolations that are splinings of curves, rather than of functions.

`spline` can handle the case when y is a vector of arbitrary specified dimensionality. The dimension can be specified with the '-d' option. For example, an input file could contain the multidimensional dataset

```
0.0  0.0  1.0
1.0  1.0  0.0
2.0  0.0  1.0
```

which are the coordinates (t, y) of the data points (0,0,1), (1,1,0), and (2,0,1). You would construct a spline (the graph of an interpolating function) passing through the points in this dataset by doing

```
spline -d 2 input_file > output_file
```

The option '-d 2' is used because in this example, the dependent variable y is a two-dimensional vector. Each of the components of y will be interpolated independently, and the output file will contain points that lie along the graph of the resulting interpolating function.

When doing multidimensional splining, you may use any of the options that apply in the default one-dimensional case. For example, the '-f' option will yield real-time cubic Bessel interpolation. As in the one-dimensional case, if the '-f' option is used then the '-t' option must be used as well, to specify an interpolation interval (a range of t values). The `-p` option will yield a periodic spline, i.e., the graph of a periodic vector-valued function. For this, the first and last dataset y values must be the same.

`spline` can also be used to draw a curve through arbitrarily chosen points in the plane, or in general through arbitrarily chosen points in d-dimensional space. This is not the same as splining, at least as the term is conventionally defined. The reason is that 'splining' refers to construction of a function, rather than the construction of a curve that may or may not be the graph of a function. Not every curve is the graph of a function.

The following example shows how you may 'spline a curve'. The command

```
echo 0 0 1 0 1 1 0 1 | spline -d 2 -a -s | graph -T X
```

will construct a curve in the plane through the four points (0,0), (1,0), (1,1), and (0,1), and graph it on an X Window System display. The '`-d 2`' option specifies that the dependent variable y is two-dimensional. The '`-a`' option specifies that t values are missing from the input, and should be automatically generated. By default, the first t value is 0, the second is 1, etc. The '`-s`' option specifies that the t values should be stripped from the output.

The same technique may be used to spline a closed curve. For example, doing

```
echo 0 0 1 0 0 1 0 0 | spline -d 2 -a -s -p | graph -T X
```

will construct and graph a closed, lozenge-shaped curve through the three points (0,0), (1,0), and (0,1). The construction of a closed curve is guaranteed by the '`-p`' (i.e., '`--periodic`') option, and by the repetition of the initial point (0,0) at the end of the sequence.

When splining a curve, whether open or closed, you may wish to substitute the '`-A`' option for the '`-a`' option. Like the '`-a`' option, the '`-A`' option specifies that t values are missing from the input and should be automatically generated. However, the increment from one t value to the next will be the distance between the corresponding values of y. This scheme for generating t values, when constructing a curve through a sequence of points, is the scheme that is used in the well known FITPACK subroutine library. It is probably the best approach when the distances between successive points fluctuate considerably.

A curve through a sequence of points in the plane, whether open or closed, may cross itself. Some interesting visual effects can be obtained by adding negative tension to such a curve. For example, doing

```
echo 0 0 1 0 1 1 0 0 | spline -d 2 -a -s -p -T -14 -n 500 | graph -T X
```

will construct a closed curve through the three points (0,0), (1,0), and (0,1), which is wound into curlicues. The '`-n 500`' option is included because there are so many windings. It specifies that 501 points should be generated, which is enough to draw a smooth curve.

7.3 `spline` command-line options

The `spline` program will interpolate vector-valued functions of a scalar variable t, and curves in d-dimensional space. The algorithms used by `spline` are similar to those discussed in D. Kincaid and [E.] W. Cheney, *Numerical Analysis* (2nd ed., Brooks/Cole, 1996), section 6.4, and C. de Boor, *A Practical Guide to Splines* (Springer-Verlag, 1978), Chapter 4.

Input file names may be specified anywhere on the command line. That is, the relative order of font names and command-line options does not matter. If no file names are specified, or the file name '`-`' is specified, the standard input is read.

An input file may contain more than a single dataset. Unless the '`-a`' or '`-A`' options are used (see below), each dataset is expected to consist of a sequence of data points, given as alternating t and y values. t is the scalar independent variable, and y is the vector-valued dependent variable. The dimensionality of y is specified with the '`-d`' option (the default is 1).

If the input file is in ASCII format (the default), its datasets are separated by blank lines. An input file may also contain any number of comment lines, which must begin with the comment character '#'. Comment lines are ignored. They are not treated as blank, i.e., they do not interrupt a dataset in progress.

The options to `spline` are listed below. There are three sorts of option:

1. Options specifying the type of interpolation to be performed on each dataset.
2. Options specifying the input or output format.
3. Options requesting information (e.g., '`--help`').

Options that take an argument are followed, in parentheses, by the type and default value of the argument.

The following options specify the type of interpolation to be performed on each dataset.

'`-f`'
'`--filter`'
: Use a local interpolation algorithm (the cubic Bessel algorithm), so that `spline` can be used as a real-time filter. The slope of the interpolating curve at each point in a dataset will be chosen by fitting a quadratic function through that point and the two adjacent points in the dataset. If '`-f`' is specified then the '`-t`' option, otherwise optional, must be used as well. Also, if '`-f`' is specified then the '`-k`', '`-p`', and '`-T`' options may not be used.

  If '`-f`' is *not* specified, then a different (global) interpolation algorithm will be used.

'`-k` *k*'
'`--boundary-condition` *k*'
: (Float, default 1.0.) Set the boundary condition parameter for each constructed spline to be k. In each of its components, the spline will satisfy the two boundary conditions $y''[0] = ky''[1]$ and $y''[n] = ky''[n-1]$. Here $y[0]$ and $y[1]$ signify the values of a specified component of the vector-valued dependent variable y at the first two points of a dataset, and $y[n-1]$ and $y[n]$ the values at the last two points. Setting k to zero will yield a 'natural' spline, i.e., one that has zero curvature at the two ends of the dataset. The '`-k`' option may not be used if '`-f`' or '`-p`' is specified.

'`-n` *n*'
'`--number-of-intervals` *n*'
: (Positive integer, default 100.) Subdivide the interval over which interpolation occurs into n subintervals. The number of data points computed, and written to the output, will be $n+1$.

'`-p`'
'`--periodic`'
: Construct a periodic spline. If this option is specified, the y values for the first and last points in each dataset must be equal. The '`-f`' and '`-k`' options may not be used if '`-p`' is specified.

'`-T` *tension*'
'`--tension` *tension*'
: (Float, default 0.0.) Set the tension in each interpolating spline to be *tension*. Between each pair of successive points in a dataset, the constructed spline will satisfy the differential equation $y'''' = \mathrm{sgn}(\mathit{tension})\,\mathit{tension}^2\, y''$ in each of its components. If *tension* equals zero, the spline will be piecewise cubic. As

tension increases to positive infinity, the spline will converge to a polygonal line. The '`-T`' option may not be used if '`-f`' is specified.

'`-t` *tmin tmax* `[`*tspacing*`]`'
'`--t-limits` *tmin tmax* `[`*tspacing*`]`'
: For each dataset, set the interval over which interpolation occurs to be the interval between *tmin* and *tmax*. If *tspacing* is not specified, the interval will be divided into the number of subintervals specified by the '`-n`' option. If the '`-t`' option is not used, the interval over which interpolation occurs will be the entire range of the independent variable in the dataset. The '`-t`' option must always be used if the '`-f`' option is used to request filter-like behavior (see above).

The following options specify the format of the input file(s) and the output file.

'`-d` *dimension*'
'`--y-dimension` *dimension*'
: (Integer, default 1.) Set the dimensionality of the dependent variable y in the input and output files to be *dimension*.

'`-I` *data-format*'
'`--input-format` *data-format*'
: (Character, default '`a`'.) Set the data format for the input file(s) to be *data-format*. The possible data formats are as follows.

'`a`' ASCII format. Each file is a sequence of floating point numbers, interpreted as the t and y coordinates of the successive data points in a dataset. If y is d-dimensional, there will be d+1 numbers for each point. The t and y coordinates of a point need not appear on the same line, and points need not appear on different lines. But if a blank line occurs (i.e., two newlines in succession are seen), it is interpreted as the end of a dataset, and the beginning of the next.

'`f`' Single precision binary format. Each file is a sequence of floating point numbers, interpreted as the t and y coordinates of the successive data points in a dataset. If y is d-dimensional, there will be $d+1$ numbers for each point. Successive datasets are separated by a single occurrence of the quantity `FLT_MAX`, which is the largest possible single precision floating point number. On most machines this is approximately 3.4×10^{38}.

'`d`' Double precision binary format. Each file is a sequence of double precision floating point numbers, interpreted as the t and y coordinates of the successive data points in a dataset. If y is d-dimensional, there will be $d+1$ numbers for each point. Successive datasets are separated by a single occurrence of the quantity `DBL_MAX`, which is the largest possible double precision floating point number. On most machines this is approximately 1.8×10^{308}.

'`i`' Integer binary format. Each file is a sequence of integers, interpreted as the t and y coordinates of the successive data points in a dataset. If y is d-dimensional, there will be $d + 1$ numbers for each point. Successive datasets are separated by a single occurrence of the quantity `INT_MAX`, which is the largest possible integer. On most machines this is $2^{31} - 1$.

'`-a [`*step_size* `[`*lower_limit*`]]`'
'`--auto-abscissa [`*step_size* `[`*lower_limit*`]]`'
: (Floats, defaults 1.0 and 0.0.) Automatically generate values for the independent variable (t). Irrespective of data format ('`a`', '`f`', '`d`', or '`i`'), this option specifies that the values of the independent variable (t) are missing from the input file: the dataset(s) to be read contain only values of the dependent variable (y), so that if y is d-dimensional, there will be only d numbers for each point. The increment from each t value to the next will be *step_size*, and the first t value will be *lower_limit*.

'`-A`'
'`--auto-dist-abscissa`'
: Automatically generate values for the independent variable (t). This is a variant form of the '`-a`' option. The increment from each t value to the next will be the distance between the corresponding y values, and the first t value will be 0.0. This option is useful when interpolating curves rather than functions (see Section 7.2 [Advanced Use of spline], page 59).

'`-O` *data-format*'
'`--output-format` *data-format*'
: (Character, default '`a`'.) Set the data format for the output file to be *data-format*. The interpretation of the *data-format* argument is the same as for the '`-I`' option.

'`-P` *significant-digits*'
'`--precision` *significant-digits*'
: (Positive integer, default 6.) Set the numerical precision for the t and y values in the output file to be *significant-digits*. This takes effect only if the output file is written in '`a`' format, i.e., in ASCII.

'`-s`'
'`--suppress-abscissa`'
: Omit the independent variable t from the output file; for each point, supply only the dependent variable y. If y is d-dimensional, there will be only d numbers for each point, not d+1. This option is useful when interpolating curves rather than functions (see Section 7.2 [Advanced Use of spline], page 59).

The following options request information.

'`--help`'
: Print a list of command-line options, and then exit.

'`--version`'
: Print the version number of `spline` and the plotting utilities package, and exit.

8 The ode Program

The GNU `ode` utility can produce a numerical solution to the initial value problem for many systems of first-order ordinary differential equations (ODE's). `ode` can also be used to solve systems of higher-order ODE's, since a simple procedure converts an n'th-order equation into n first-order equations. The output of `ode` can easily be piped to `graph`, so that one or more solution curves may be plotted as they are generated.

Three distinct schemes for numerical solution are implemented: Runge–Kutta–Fehlberg (the default), Adams–Moulton, and Euler. The Runge–Kutta–Fehlberg and Adams–Moulton schemes are available with adaptive stepsize.

8.1 Mathematical basics

We begin with some standard definitions. A *differential equation* is an equation involving an unknown function and its derivatives. A differential equation is *ordinary* if the unknown function depends on only one independent variable, often denoted t. The *order* of the differential equation is the order of the highest-order derivative in the equation. One speaks of a family, or *system* of equations when more than one equation is involved. If the equations are dependent on one another, they are said to be *coupled*. A *solution* is any function satisfying the equations. An *initial value problem* is present when there exist subsidiary conditions on the unknown function and its derivatives, all of which are given at the same value of the independent variable. In principle, such an 'initial condition' specifies a unique solution. Questions about the existence and uniqueness of a solution, along with further terminology, are discussed in any introductory text. (See Chapter 1 of Birkhoff and Rota's *Ordinary Differential Equations*. For this and other references relevant to `ode`, see Section 8.9 [ODE Bibliography], page 81.)

In practical problems, the solution of a differential equation is usually not expressible in terms of elementary functions. Hence the need for a numerical solution.

A numerical scheme for solving an initial value problem produces an approximate solution, using only functional evaluations and the operations of arithmetic. `ode` solves first-order initial value problems of the form:

$$
\begin{aligned}
x' &= f(t,x,y,\ldots,z)\\
y' &= g(t,x,y,\ldots,z)\\
&\ \vdots\\
z' &= h(t,x,y,\ldots,z)
\end{aligned}
$$

given the initial values for each dependent variable at the initial value of the independent variable t, i.e.,

$$
\begin{aligned}
x(a) &= b\\
y(a) &= c\\
&\ \vdots\\
z(a) &= d\\
t &= a
\end{aligned}
$$

where $a, b, c, \ldots, d$ are constants.

For ode to be able to solve such a problem numerically, the functions $f, g, \ldots, h$ must be expressed, using the usual operators (plus, minus, multiplication, division, and exponentiation), in terms of certain basic functions that ode recognizes. These are the same functions that the plotting program gnuplot recognizes. Moreover, each of $f, g, \ldots, h$ must be given explicitly. ode cannot deal with a system in which one or more of the first derivatives is defined implicitly rather than explicitly.

All schemes for numerical solution involve the calculation of an approximate solution at discrete values of the independent variable t, where the 'stepsize' (the difference between any two successive values of t, usually denoted h) may be constant or chosen adaptively. In general, as the stepsize decreases the solution becomes more accurate. In ode, the stepsize can be adjusted by the user.

8.2 Simple examples using ode

The following examples should illustrate the procedure of stating an initial value problem and solving it with ode. If these examples are too elementary, see Section 8.8 [Input Language], page 78, for a formal specification of the ode input language. There is also a directory containing examples of ode input, which is distributed along with the GNU plotting utilities. On most systems it is installed as '/usr/share/ode' or '/usr/local/share/ode'.

Our first example is a simple one, namely

$$y'(t) = y(t)$$

with the initial condition

$$y(0) = 1$$

The solution to this differential equation is

$$y(t) = e^t .$$

In particular

$$y(1) = e^1 = 2.718282$$

to seven digits of accuracy.

You may obtain this result with the aid of ode by typing on the command line the sequence of commands

```
ode
y' = y
y = 1
print t, y
step 0, 1
```

Two columns of numbers will appear. Each line will show the value of the independent variable t, and the value of the variable y, as t is 'stepped' from 0 to 1. The last line will be

```
1 2.718282
```

as expected. You may use the '-p' option to change the precision. If, for example, you type 'ode -p 10' rather than 'ode', you will get ten digits of accuracy in the output, rather than seven (the default).

After the above output, `ode` will wait for further instructions. Entering for example the line

```
step 1, 0
```

should yield two more columns of numbers, containing the values of t and y that are computed when t is stepped back from 1 to 0. You could type instead

```
step 1, 2
```

to increase rather than decrease t. To exit `ode`, you would type a line containing only '`.`', i.e. a single period, and tap 'return'. `ode` will also exit if it sees an end-of-file indicator in its input stream, which you may send from your terminal by typing control-D.

Each line of the preceding example should be self-explanatory. A '`step`' statement sets the beginning and the end of an interval over which the independent variable (here, t) will range, and causes `ode` to set the numerical scheme in motion. The initial value appearing in the first '`step`' statement (i.e., 0) and the assignment statement

```
y = 1
```

are equivalent to the initial condition $y(0) = 1$. The statements '`y' = y`' and '`y = 1`' are very different: '`y' = y`' defines a way of computing the derivative of y, while '`y = 1`' sets the initial value of y. Whenever a '`step`' statement is encountered, `ode` tries to step the independent variable through the interval it specifies. Which values are to be printed at each step is specified by the most recent '`print`' statement. For example,

```
print t, y, y'
```

would cause the current value of the independent variable t, the variable y, and its derivative to be printed at each step.

To illustrate `ode`'s ability to take its input or the initial part of its input from a file, you could prepare a file containing the following lines:

```
# an ode to Euler
y  = 1
y' = y
print t, y, y'
```

Call this file '`euler`'. (The '`#`' line is a comment line, which may appear at any point. Everything from the '`#`' to the end of the line on which it appears will be ignored.) To process this file with `ode`, you could type on your terminal

```
ode -f euler
step 0, 1
```

These two lines cause `ode` to read the file '`euler`', and the stepping to take place. You will now get three quantities (t, y, and y') printed at each of the values of t between 0 and 1. At the conclusion of the stepping, `ode` will wait for any further commands to be input from the terminal. This example illustrates that

```
ode -f euler
```

is not equivalent to

```
ode < euler
```

The latter would cause `ode` to take all its input from the file '`euler`', while the former allows subsequent input from the terminal. For the latter to produce output, you would need to include a '`step`' line at the end of the file. You would not need to include a '`.`' line, however. '`.`' is used to terminate input only when input is being read from a terminal.

A second simple example involves the numerical solution of a second-order differential equation. Consider the initial value problem

$$\begin{aligned} y''(t) &= -y(t) \\ y(0) &= 0 \\ y'(0) &= 1 \end{aligned}$$

Its solution would be

$$y(t) = \sin(t)$$

To solve this problem using `ode`, you must express this second-order equation as two first-order equations. Toward this end you would introduce a new function, called yp say, of the independent variable t. The pair of equations

$$\begin{aligned} y' &= yp \\ yp' &= -y \end{aligned}$$

would be equivalent to the single equation above. This sort of reduction of an n'th order problem to n first order problems is a standard technique.

To plot the variable y as a function of the variable t, you could create a file containing the lines

```
# sine : y''(t) = -y(t), y(0) = 0, y'(0) = 1
sine' = cosine
cosine' = -sine
sine = 0
cosine = 1
print t, sine
```

(y and yp have been renamed *sine* and *cosine*, since that is what they will be.) Call this file '`sine`'. To display the generated data points on an X Window System display as they are generated, you would type

```
ode -f sine | graph -T X -x 0 10 -y -1 1
step 0, 2*PI
.
```

After you type the `ode` line, `graph -T X` will pop up a window, and after you type the '`step`' line, the generated dataset will be drawn in it. The '`-x 0 10`' and '`-y -1 1`' options, which set the bounds for the two axes, are necessary if you wish to display points in real time: as they are generated. If the axis bounds were not specified on the command line, `graph -T X` would wait until all points are read from the input before determining the bounds, and drawing the plot.

A slight modification of this example, showing how `ode` can generate several datasets in succession and plot them on the same graph, would be the following. Suppose that you type on your terminal the following lines.

```
ode -f sine | graph -T X -C -x 0 10 -y -1 1
step 0, PI
step PI, 2*PI
step 2*PI, 3*PI
.
```

Then the sine curve will be traced out in three stages. Since the output from each '`step`' statement ends with a blank line, `graph -T X` will treat each section of the sine curve as a different dataset. If you are using a color display, each of the three sections will be plotted in a different color. This is a feature provided by `graph`, which normally changes its linemode after each dataset it reads. If you do not like this feature, you may turn it off by using `graph -T X -B` instead of `graph -T X`.

In the above examples, you could use any of the other variants of `graph` instead of `graph -T X`. For example, you could use `graph -T ps` to obtain a plot in encapsulated Postscript format, by typing

```
ode -f sine | graph -T ps > plot.ps
step 0, 2*PI
.
```

You should note that of the variants of `graph`, the seven variants `graph -T pnm`, `graph -T gif`, `graph -T ai`, `graph -T ps`, `graph -T fig`, `graph -T pcl` and `graph -T hpgl` do not produce output in real time, even when the axis bounds are specified with the '`-x`' and '`-y`' options. So if any of these seven variants is used, the plot will be produced only when input from `ode` is terminated, which will occur when you type '`.`'.

In the preceding examples, the derivatives of the dependent variables were specified by comparatively simple expressions. They are allowed to be arbitrarily complicated functions of the dependent variables and the independent variable. They may also involve any of the functions that are built into `ode`. `ode` has a fair number of functions built in, including `abs`, `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, and `atanh`. Less familiar functions which are built into it are `besj0`, `besj1`, `besy0`, `besy1`, `erf`, `erfc`, `inverf`, `lgamma`, `gamma`, `norm`, `invnorm`, `ibeta`, and `igamma`. These have the same definitions as in the plotting program `gnuplot`. (All functions take a single argument, except for `ibeta`, which takes three, and `igamma`, which takes two). `ode` also knows the meaning of the constant '`PI`', as the above examples show. The names of the preceding functions are reserved, so, e.g., '`cos`' and '`sin`' may not be used as names for variables.

Other than the restriction of avoiding reserved names and keywords, the names of variables may be chosen arbitrarily. Any sequence of alphanumeric characters starting with an alphabetic character may be used; the first 32 characters are significant. It is worth noting that `ode` identifies the independent variable by the fact that it is (or should be) the only variable that has not appeared on the left side of a differential equation or an initial value assignment. If there is more than than one such variable then no stepping takes place; instead, an error message is printed. If there is no such variable, a dummy independent variable is invented and given the name '`(indep)`', internally.

8.3 Additional examples using `ode`

We explain here how to use some additional features of `ode`. However, the discussion below does not cover all of its capabilities. For a complete list of command-line options, see Section 8.4 [ode Invocation], page 71.

It is easy to use `ode` to create plots of great beauty. An example would be a plot of a *strange attractor*, namely the Lorenz attractor. Suppose that a file named '`lorenz`' contains the following lines.

```
# The Lorenz model, a system of three coupled ODE's with parameter r.
x' = -3*(x-y)
y' = -x*z+r*x-y
z' = x*y-z
r = 26
x = 0; y = 1; z = 0
print x, y
step 0, 200
```

Then executing the command

```
ode < lorenz | graph -T X -C -x -10 10 -y -10 10
```

would produce a plot of the Lorenz attractor (strictly speaking, a plot of one of its two-dimensional projections). You may produce a Postscript plot of the Lorenz attractor, and print it, by doing something like

```
ode < lorenz | graph -T ps -x -10 10 -y -10 10 -W 0 | lpr
```

The '`-W 0`' ("zero width") option requests that `graph -T ps` use the thinnest line possible, to improve the visual appearance of the plot on a printer or other Postscript device.

Besides plotting a visually striking object in real time, the Lorenz attractor example shows how statements may be separated by semicolons, rather than appearing on different lines. It also shows how to use symbolic constants. In the description read by `ode` the parameter r is a variable like x, y, and z. But unlike them it is not updated during stepping, since no formula for its derivative r' is given.

Our second example deals with the interactive construction of a 'phase portrait': a set of solution curves with different initial conditions. Phase portraits are of paramount interest in the qualitative theory of differential equations, and also possess æsthetic appeal.

Since a description read by `ode` may contain any number of '`step`' statements, multiple solution curves may be plotted in a single run. The most recent '`print`' statement will be used with each '`step`' statement. In practice, a phase portrait would be drawn from a few well-chosen solution curves. Choosing a good set of solution curves may require experimentation, which makes interactivity and real-time plotting all-important.

As an example, consider a so-called Lotka–Volterra predator–prey model. Suppose that in a lake there are two species of fish: A (the prey) who live by eating a plentiful supply of plants, and B (the predator) who eat A. Let $x(t)$ be the population of A and $y(t)$ the population of B at time t. A crude model for the interaction of A and B is given by the equations

$$\begin{aligned} x' &= x(a - by) \\ y' &= y(cx - d) \end{aligned}$$

where a, b, c, d are positive constants. To draw a phase portrait for this system interactively, you could type

```
ode | graph -T X -C -x 0 5 -y 0 5
x' = (a - b*y) * x
y' = (c*x - d) * y
a = 1; b = 1; c = 1; d = 1;
print x, y
```

```
x = 1; y = 2
step 0, 10
x = 1; y = 3
step 0, 10
x = 1; y = 4
step 0, 10
x = 1; y = 5
step 0, 10
.
```

Four curves will be drawn in succession, one per '`step`' line. They will be periodic; this periodicity is similar to the fluctuations between predator and prey populations that occur in real-world ecosystems. On a color display the curves will appear in different colors, since by default, `graph` changes the linemode between datasets. That feature may be turned off by using `graph -T X -B` rather than `graph -T X`.

It is sometimes useful to use `ode` and `graph` to plot discrete points, which are not joined by line segments to form a curve. Our third example illustrates this. Suppose the file '`atwoods`' contains the lines

```
m = 1
M = 1.0625
a = 0.5; adot = 0
l = 10; ldot = 0

ldot' = ( m * l * adot * adot - M * 9.8 + m * 9.8 * cos(a) ) / (m + M)
l'    = ldot
adot' = (-1/l) * (9.8 * sin(a) +  2 * adot * ldot)
a'    = adot

print l, ldot
step 0, 400
```

The first few lines describe the functioning of a so-called swinging Atwood's machine. An ordinary Atwood's machine consists of a taut cord draped over a pulley, with a mass attached to the cord at each end. Normally, the heavier mass (M) would win against the lighter mass (m), and draw it upward. A swinging Atwood's machine allows the lighter mass to swing back and forth as well as move vertically.

The '`print l, ldot`' statement requests that the vertical position and vertical velocity of the lighter mass be printed out at each step. If you run the command

```
ode < atwoods | graph -T X -x 9 11 -y -1 1 -m 0 -S 1 -X l -Y ldot
```

you will obtain a real-time plot. The '`-m 0`' option requests that successive data points not be joined by line segments, and the '`-S 1`' option requests that plotting symbol #1 (a dot) be plotted at the location of each point. As you will see if you run this command, the heavy mass does not win against the lighter mass. Instead the machine oscillates non-periodically. Since the motion is non-periodic, the plot benefits from being drawn as a sequence of unconnected points.

We conclude by mentioning a few features of `ode` that may be useful when things are not going quite right. One of them is the '`examine`' statement. It may be used to discover pertinent information about any variable in a system. For details, see Section 8.8 [Input Language], page 78.

Another useful feature is that the '`print`' statement may be used to print out more than just the value of a variable. As we have seen, if the name of the variable is followed by '`'`', the derivative of the variable will be printed instead. In a similar way, following the variable name with '`?`', '`!`', or '`~`' prints respectively the relative single-step error, the absolute single-step error, or the accumulated error (not currently implemented). These quantities are discussed in Section 8.6 [Numerical Error], page 74.

The '`print`' statement may be more complicated than was shown in the preceding examples. Its general structure is

```
print <pr-list> [every <const>] [from <const>]
```

The bracket notation '`[...]`' means that the enclosed statements are optional. Until now we have not mentioned the '`every`' clause or the '`from`' clause. The `<pr-list>` is familiar, however; it is simply a comma-separated list of variables. For example, in the statement

```
print t, y, y' every 5 from 1
```

the `<pr-list>` is `<t, y, y'>`. The clauses '`every 5`' and '`from 1`' specify that printing should take place after every fifth step, and that the printing should begin when the independent variable t reaches 1. An '`every`' clause is useful if you wish to 'thin out' the output generated by a '`step`' statement, and a '`from`' clause is useful if you wish to view only the final portion of a solution curve.

8.4 ode command-line options

The command-line options to `ode` are listed below. There are several sorts of option:

1. Options affecting the way in which input is read.
2. Options affecting the format of the output.
3. Options affecting the choice of numerical solution scheme, and the error bounds that will be imposed on it.
4. Options that request information.

The following option affects the way input is read.

'`-f` *filename*'
'`--input-file` *filename*'
: Read input from *filename* before reading from standard input.

The following options affect the output format.

'`-p` *significant-digits*'
'`--precision` *significant-digits*'
: (Positive integer, default 6.) When printing numerical results, use a precision specified by *significant-digits*. If this option is given, the print format will be scientific notation.

'`-t`'
'`--title`'
: Print a title line at the head of the output, naming the columns. If this option is given, the print format will be scientific notation.

The following options specify the numerical integration scheme. Only one of the three basic option '`-R`', '`-A`', and '`-E`' may be specified. The default is '`-R`' (Runge–Kutta–Fehlberg).

'-R [*stepsize*]'
'--runge-kutta [*stepsize*]'
Use a fifth-order Runge–Kutta–Fehlberg algorithm, with an adaptive stepsize unless a constant stepsize is specified. When a constant stepsize is specified and no error analysis is requested, then a classical fourth-order Runge–Kutta scheme is used.

'-A [*stepsize*]'
'--adams-moulton [*stepsize*]'
Use a fourth-order Adams–Moulton predictor–corrector scheme, with an adaptive stepsize unless a constant stepsize, *stepsize*, is specified. The Runge–Kutta–Fehlberg algorithm is used to get past 'bad' points (if any).

'-E [*stepsize*]'
'--euler [*stepsize*]'
Use a 'quick and dirty' Euler scheme, with a constant stepsize. The default value of *stepsize* is 0.1. Not recommended for serious applications.

The error bound options '-r' and '-e' (see below) may not be used if '-E' is specified.

'-h *hmin* [*hmax*]'
'--step-size-bound *hmin* [*hmax*]'
Use a lower bound *hmin* on the stepsize. The numerical scheme will not let the stepsize go below *hmin*. The default is to allow the stepsize to shrink to the machine limit, i.e., the minimum nonzero double-precision floating point number. The optional argument *hmax*, if included, specifies a maximum value for the stepsize. It is useful in preventing the numerical routine from skipping quickly over an interesting region.

The following options set the error bounds on the numerical solution scheme.

'-r *rmax* [*rmin*]'
'--relative-error-bound *rmax* [*rmin*]'
'-e *emax* [*emin*]'
'--absolute-error-bound *emax* [*emin*]'
The '-r' option sets an upper bound on the relative single-step error. If the '-r' option is used, the relative single-step error in any dependent variable will never exceed *rmax* (the default for which is 10^{-9}). If this should occur, the solution will be abandoned and an error message will be printed. If the stepsize is not constant, the stepsize will be decreased 'adaptively', so that the upper bound on the single-step error is not violated. Thus, choosing a smaller upper bound on the single-step error will cause smaller stepsizes to be chosen. A lower bound *rmin* may optionally be specified, to suggest when the stepsize should be increased (the default for *rmin* is *rmax*/1000). The '-e' option is similar to '-r', but bounds the absolute rather than the relative single-step error.

'-s'
'--suppress-error-bound'
Suppress the ceiling on single-step error, allowing ode to continue even if this ceiling is exceeded. This may result in large numerical errors.

Finally, the following options request information.

'--help' Print a list of command-line options, and then exit.

'--version'
Print the version number of ode and the plotting utilities package, and exit.

8.5 Diagnostic messages

ode is always in one of two states:

- Reading input. The input includes a specification of a system of ordinary differential equations, together with instructions for solving it numerically: a 'print' line and a 'step' line.
- Numerically solving a system, and printing the resulting output.

ode moves from the first to the second state after it sees and processes a 'step' line. It returns to the first state after the generated output has been printed. Errors may occur in either the 'reading' state or the 'solving' state, and may terminate computations or even cause ode to exit. We now explain the possible sorts of error.

While reading input, ode may encounter a syntax error: an ungrammatical line that it is unable to parse. (For a summary of its input grammar, see Section 8.8 [Input Language], page 78.) If so, it emits the error message

```
ode::nnn: syntax error
```

where 'nnn' is the number of the line containing the error. When the '-f filename' option is used to specify an input file, the error message will read

```
ode:filename:nnn: syntax error
```

for errors encountered inside the input file. Subsequently, when ode begins reading the standard input, line numbers will start over again from 1.

No effort is made to recover from syntax errors in the input. However, there is a meager effort to resynchronize, so that more than one syntax error in a file may be found at the same time.

It is also possible that a fatal arithmetic exception (such as a division by zero, or a floating point overflow) may occur while ode is reading input. If such an exception occurs, ode will print an "Floating point exception" error message and exit. Arithmetic exceptions are machine-dependent. On some machines, the line

```
y = 1/0
```

would induce an arithmetic exception. Also on some machines (not necessarily the same ones), the lines

```
y = 1e100
z = y^4
```

would induce an arithmetic exception. That is because on most machines, the double precision quantities that ode uses internally are limited to a maximum size of approximately 1.8×10^{308}.

When ode is in the 'solving' state, i.e., computing a numerical solution, similar arithmetic exceptions may occur. If so, the solution will be interrupted and a message resembling

```
ode: arithmetic exception while calculating y'
```

will be printed. However, ode will not exit; the exception will be 'caught'. ode itself recognizes the following exceptional conditions: square root of a negative number, logarithm of a non-positive number, and negative number raised to a non-integer power. ode will catch any of these operations before it is performed, and print an error message specifying which illegal operation it has encountered.

```
ode: square root of a negative number while calculating y'
```

would be a typical error message.

If the machine on which ode is running supports the 'matherr' facility for reporting errors in the computation of standard mathematical functions, it will be used. This facility reports domain errors and range errors (overflows, underflows, and losses of significance) that could occur when evaluating such functions as 'log', 'gamma', etc.; again, before they are performed. If the matherr facility is present, the error message will be fairly informative. For example, the error message

```
ode: range error (overflow) in lgamma while calculating y'
```

could be generated if the logarithmic gamma function 'lgamma' is evaluated at a value of its argument that is too large. The generation of any such message, except a message warning of an underflow, will cause the numerical solution to be interrupted.

There is another sort of error that may occur during numerical solution: the condition that an error ceiling, which the user may set with the '-r' option or the '-e' option, is exceeded. This too will cause the numerical solution to be abandoned, and ode to switch back to reading input.

8.6 Numerical error and how to avoid it

This discussion is necessarily incomplete. Entire books exist on any subject mentioned below (e.g., floating point error). Our goals are modest: first, to introduce the basic notions of error analysis as they apply to ode; second, to steer you around the more obvious pitfalls. You should look through a numerical analysis text (e.g., Atkinson's *Introduction to Numerical Analysis*) before beginning this discussion.

We begin with some key definitions. The error of greatest concern is the difference between the actual solution and the numerical approximation to the solution; this is termed the *accumulated error*, since the error is built up during each numerical step. Unfortunately, an estimate of this error is usually not available without knowledge of the actual solution. There are, however, several more usable notions of error. The *single-step error*, in particular, is the difference between the actual solution and the numerical approximation to the solution after any single step, assuming the value at the beginning of the step is correct.

The *relative single-step error* is the single-step error, divided by the current value of the numerical approximation to the solution. Why not divided by the current value of the solution itself? The reason is that the solution is not exactly known. When free to choose a stepsize, ode will do so on the basis of the relative single-step error. By default, it will choose the stepsize so as to maintain an accuracy of eight significant digits in each step. That is, it will choose the stepsize so as not to violate an upper bound of 10^{-9} on the relative single-step error. This ceiling may be adjusted with the '-r' option.

Where does numerical error come from? There are two sources. The first is the finite precision of machine computation. All computers work with floating point numbers, which are not real numbers, but only an approximation to real numbers. However, all computations performed by `ode` are done to double precision, so floating point error tends to be relatively small. You may nonetheless detect the difference between real numbers and floating point numbers by experimenting with the '`-p 17`' option, which will print seventeen significant digits. On most machines, that is the precision of a double precision floating point number.

The second source of numerical error is often called the *theoretical truncation error*. It is the difference between the actual solution and the approximate solution due solely to the numerical scheme. At the root of many numerical schemes is an infinite series; for ordinary differential equations, it is a Taylor expansion. Since the computer cannot compute all the terms in an infinite series, a numerical scheme necessarily uses a truncated series; hence the term. The single-step error is the sum of the theoretical truncation error and the floating point error, though in practice the floating point error is seldom included. The single-step error estimated by `ode` consists only of the theoretical truncation error.

We say that a numerical scheme is *stable*, when applied to a particular initial value problem, if the error accumulated during the solution of the problem over a fixed interval decreases as the stepsize decreases; at least, over a wide range of step sizes. With this definition both the Runge–Kutta–Fehlberg ('`-R`') scheme and the Adams–Moulton ('`-A`') scheme are stable (a statement based more on experience than on theoretical results) for a wide class of problems.

After these introductory remarks, we list some common sources of accumulated error and instability in any numerical scheme. Usually, problems with large accumulated error and instability are due to the single-step error in the vicinity of a 'bad' point being large.

1. Singularities.

   `ode` should not be used to generate a numerical solution on any interval containing a singularity. That is, `ode` should not be asked to step over points at which the system of differential equations is singular or undefined.

   You will find the definitions of singular point, regular singular point, and irregular singular point in any good differential equations text. If you have no favorite, try Birkhoff and Rota's *Ordinary Differential Equations*, Chapter 9. Always locate and classify the singularities of a system, if any, before applying `ode`.

2. Ill-posed problems.

   For `ode` to yield an accurate numerical solution on an interval, the true solution must be defined and well-behaved on that interval. The solution must also be real. Whenever any of these conditions is violated, the problem is said to be *ill-posed*. Ill-posedness may occur even if the system of differential equations is well-behaved on the interval. Strange results, e.g., the stepsize suddenly shrinking to the machine limit or the solution suddenly blowing up, may indicate ill-posedness.

   As an example of ill-posedness (in fact, an undefined solution) consider the innocent-looking problem:

   $$\begin{aligned} y' &= y^2 \\ y(1) &= -1 \end{aligned}$$

   The solution on the domain $t>0$ is

$$y(t) \;=\; -1/t\,.$$

With this problem you must not compute a numerical solution on any interval that includes $t = 0$. To convince yourself of this, try to use the '`step`' statement

```
step 1, -1
```

on this system. How does `ode` react?

As another example of ill-posedness, consider the system

$$y' = 1/y$$

which is undefined at $y = 0$. The general solution is

$$y \;=\; \pm\;(2(t-C))^{1/2},$$

so that if the condition $y(2) = 2$ is imposed, the solution will be $(2t)^{1/2}$. Clearly, if the domain specified in a '`step`' statement includes negative values of t, the generated solution will be bogus.

In general, when using a constant stepsize you should be careful not to 'step over' bad points or bad regions. When allowed to choose a stepsize adaptively, `ode` will often spot bad points, but not always.

3. Critical points.

   An *autonomous* system is one that does not include the independent variable explicitly on the right-hand side of any differential equation. A *critical point* for such a system is a point at which all right-hand sides equal zero. For example, the system

   $$\begin{aligned} y' &= 2x \\ x' &= 2y \end{aligned}$$

   has only one critical point, at $(x, y) = (0, 0)$.

   A critical point is sometimes referred to as a *stagnation point*. That is because a system at a critical point will remain there forever, though a system near a critical point may undergo more violent motion. Under some circumstances, passing near a critical point may give rise to a large accumulated error.

   As an exercise, solve the system above using `ode`, with the initial condition $x(0) = y(0) = 0$. The solution should be constant in time. Now do the same with points near the critical point. What happens?

   You should always locate the critical points of a system before attempting a solution with `ode`. Critical points may be classified (as equilibrium, vortex, unstable, stable, etc.) and this classification may be of use. To find out more about this, consult any book dealing with the qualitative theory of differential equations (e.g., Birkhoff and Rota's *Ordinary Differential Equations*, Chapter 6).

4. Unsuitable numerical schemes

   If the results produced by `ode` are bad in the sense that instability appears to be present, or an unusually small stepsize needs to be chosen needed in order to reduce the single-step error to manageable levels, it may simply be that the numerical scheme being used is not suited to the problem. For example, `ode` currently has no numerical scheme which handles so-called 'stiff' problems very well.

   As an example, you may wish to examine the stiff problem:

   $$\begin{aligned} y' &= -100 + 100t + 1 \\ y(0) &= 1 \end{aligned}$$

on the domain $[0, 1]$. The exact solution is

$$y(t) \;=\; e^{-100t} \;+\; t\,.$$

It is a useful exercise to solve this problem with `ode` using various numerical schemes, stepsizes, and relative single-step error bounds, and compare the generated solution curves with the actual solution.

There are several rough and ready heuristic checks you may perform on the accuracy of any numerical solution produced by `ode`. We discuss them in turn.

1. Examine the stability of solution curves: do they converge?

   That is, check how changing the stepsize affects a solution curve. As the stepsize decreases, the curve should converge. If it does not, then either the stepsize is not small enough or the numerical scheme is not suited to the problem. In practice, you would proceed as follows.

   - If using an adaptive stepsize, superimpose the solution curves for successively smaller bounds on the relative single-step error (obtained with, e.g., '`-r 1e-9`', '`-r 1e-11`', '`-r 1e-13`', ...). If the curves converge then the solution is to all appearances stable, and your accuracy is sufficient.
   - If employing a constant stepsize, perform a similar analysis by successively halving the stepsize.

   The following example is one that you may wish to experiment with. Make a file named '`qcd`' containing:

   ```
   # an equation arising in QCD (quantum chromodynamics)
   f'   = fp
   fp'  = -f*g^2
   g'   = gp
   gp'  = g*f^2
   f = 0; fp = -1; g = 1; gp = -1

   print t, f
   step 0, 5
   ```

   Next make a file named '`stability`', containing the lines:

   ```
   : sserr is the bound on the relative single-step error
   for sserr
   do
   ode -r $sserr < qcd
   done | spline -n 500 | graph -T X -C
   ```

   This is a 'shell script', which when run will superimpose numerical solutions with specified bounds on the relative single-step error. To run it, type:

   ```
   sh stability 1 .1 .01 .001
   ```

   and a plot of the solutions with the specified error bounds will be drawn. The convergence, showing stability, should be quite illuminating.

2. Check invariants of the system: are they constant?

   Many systems have invariant quantities. For example, if the system is a mathematical model of a 'conservative' physical system then the 'energy' (a particular function of the dependent variables of the system) should be constant in time. In general, knowledge about the qualitative behavior of any dependent variable may be used to check the quality of the solution.

3. Check a family of solution curves: do they diverge?

   A rough idea of how error is propagated is obtained by viewing a family of solution curves about the numerical solution in question, obtained by varying the initial conditions. If they diverge sharply—that is, if two solutions which start out very close nonetheless end up far apart—then the quality of the numerical solution is dubious. On the other hand, if the curves do not diverge sharply then any error that is present will in all likelihood not increase by more than an order of magnitude or so over the interval. Problems exhibiting no sharp divergence of neighboring solution curves are sometimes called *well-conditioned*.

8.7 Running time

The time required for `ode` to solve numerically a system of ordinary differential equations depends on a great many factors. A few of them are: number of equations, complexity of equations (number of operators and nature of the operators), and number of steps taken (a very complicated function of the difficulty of solution, unless constant stepsizes are used). The most effective way to gauge the time required for solution of a system is to clock a short or imprecise run of the problem, and reason as follows: the time required to take two steps is roughly twice that required for one; and there is a relationship between the number of steps required and the relative error ceiling chosen. That relationship depends on the numerical scheme being used, the difficulty of solution, and perhaps on the magnitude of the error ceiling itself. A few carefully planned short runs may be used to determine this relationship, enabling a long but imprecise run to be used as an aid in projecting the cost of a more precise run over the same region. Lastly, if a great deal of data is printed, it is likely that more time is spent in printing the results than in computing the numerical solution.

8.8 The `ode` input language formally specified

The following is a formal specification of the grammar for `ode`'s input language, in Backus–Naur form. Nonterminal symbols in the grammar are enclosed in angle brackets. Terminal tokens are in all capitals. Bare words and symbols stand for themselves.

```
<program>    ::=        ... empty ...
                 |  <program> <statement>


<statement>  ::=  SEP
                 |  IDENTIFIER = <const> SEP
                 |  IDENTIFIER ' = <expression> SEP
                 |  print <printlist> <optevery> <optfrom> SEP
                 |  step <const> , <const> , <const> SEP
                 |  step <const> , <const> SEP
                 |  examine IDENTIFIER SEP


<printlist>  ::=  <printitem>
                 |  <printlist> , <printitem>
```

```
<printitem>  ::=  IDENTIFIER
                |  IDENTIFIER '
                |  IDENTIFIER ?
                |  IDENTIFIER !
                |  IDENTIFIER ~

<optevery>   ::=        ... empty ...
                |  every <const>

<optfrom>    ::=        ... empty ...
                |  from <const>

<const>      ::=  <expression>


<expression> ::=  ( <expression> )
                |  <expression> + <expression>
                |  <expression> - <expression>
                |  <expression> * <expression>
                |  <expression> / <expression>
                |  <expression> ^ <expression>
                |  FUNCTION ( <expression> )
                |  - <expression>
                |  NUMBER
                |  IDENTIFIER
```

Since this grammar is ambiguous, the following table summarizes the precedences and associativities of operators within expressions. Precedences decrease from top to bottom.

Class	Operators	Associativity
Exponential	^	right
Multiplicative	* /	left
Additive	+ -	left

As noted in the grammar, there are six types of nontrivial statement. We now explain the effects (the 'semantics') of each type, in turn.

1. `IDENTIFIER ' = <expression>`

   This defines a first-order differential equation. The derivative of `IDENTIFIER` is specified by `<expression>`. If a dynamic variable does not appear on the left side of a statement of this form, its derivative is assumed to be zero. That is, it is a symbolic constant.

2. `IDENTIFIER = <const>`

   This sets the value of `IDENTIFIER` to the current value of `<expression>`. Dynamic variables that have not been initialized in this way are set to zero.

3. `step <const> , <const>`

4. step <const> , <const> , <const>

   A ‘step’ statement causes the numerical scheme to be executed. The first <const> is the initial value of the independent variable. The second is its final value. The third is a stepsize; if given, it overrides any stepsize that may be specified on the command line. Usually the stepsize is not specified, and it varies adaptively as the computation proceeds.

5. print <printlist> [every <const>] [from <const>]

   A ‘print’ statement controls the content and frequency of the numerical output. <printlist> is a comma-separated list of IDENTIFIERs, where each IDENTIFIER may be followed by ‘'’, denoting the derivative, or ‘?’, denoting the relative single-step error, or ‘!’, denoting the absolute single-step error, or ‘~’, denoting the accumulated error (not currently implemented). The specified values are printed in the order they are found. Both the ‘every’ clause and the ‘from’ clause are optional. If the ‘every’ clause is present, a printing occurs every <const> iterations of the numerical algorithm. The default is to print on every iteration (i.e. ‘every 1’). The first and last values are always printed. If the ‘from’ clause is present, it means to begin printing when the independent variable reaches or exceeds <const>. The default is to begin printing immediately.

   If no ‘print’ statement has been supplied, then the independent variable and all dependent variables which have differential equations associated with them are printed. The independent variable is printed first; the dependent variables follow in the order their equations were given.

6. examine IDENTIFIER

   An ‘examine’ statement, when executed, causes a table of interesting information about the named variable to be printed on the standard output. For example, if the statement ‘examine y’ were encountered after execution of the ‘ode to Euler’ example discussed elsewhere, the output would be:

   ```
   "y" is a dynamic variable
   value:2.718282
   prime:2.718282
   sserr:1.121662e-09
   aberr:3.245638e-09
   acerr:0
    code:  push "y"
   ```

   The phrase ‘dynamic variable’ means that there is a differential equation describing the behavior of y. The numeric items in the table are:

   value — Current value of the variable.

   prime — Current derivative of the variable.

   sserr — Relative single-step error for the last step taken.

   aberr — Absolute single-step error for the last step taken.

   acerr — Total error accumulated during the most recent ‘step’ statement. Not currently implemented.

The 'code' section of the table lists the stack operations required to compute the derivative of y (somewhat reminiscent of a reverse Polish calculator). This information may be useful in discovering whether the precedences in the differential equation statement were interpreted correctly, or in determining the time or space expense of a particular calculation. 'push "y"' means to load y's value on the stack, which is all that is required to compute its derivative in this case.

The grammar for the ode input language contains four types of terminal token: FUNCTION, IDENTIFIER, NUMBER, and SEP. They have the following meanings.

1. FUNCTION

   One of the words: abs, sqrt, exp, log, ln, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, floor, ceil, besj0, besj1, besy0, besy1, erf, erfc, inverf, lgamma, gamma, norm, invnorm, ibeta, igamma. These are defined to have the same meaning as in the plotting program gnuplot. All functions take a single argument, except for ibeta, which takes three, and igamma, which takes two. For trigonometric functions, all arguments are expressed in radians. The atan function is defined to give a value between −PI/2 and PI/2 (inclusive).

2. IDENTIFIER

   A sequence of alphanumeric characters starting with an alphabetic character. The first 32 characters are significant. Upper and lower-case letters are distinct. In identifiers, the underscore character is considered alphabetic. Function names and keywords may not be used as identifiers, nor may 'PI'.

3. NUMBER

   A non-empty sequence of digits possibly containing a decimal point and possibly followed by an exponent. An exponent is 'e' or 'E', followed by an (optionally signed) one, two, or three-digit number. All numbers and all parts of numbers are radix 10. A number may not contain any white space. The special word 'PI' is a number.

4. SEP

   A separator: a semicolon or a (non-escaped) newline.

In the ode input language, upper and lower-case letters are distinct. Comments begin with the character '#' and continue to the end of the line. Long lines may be continued onto a second line by ending the first line with a backslash ('\'). That is because the combination backslash-newline is equivalent to a space.

Spaces or tabs are required in the input whenever they are needed to separate identifiers, numbers, and keywords from one another. Except as separators, they are ignored.

8.9 Bibliography on ode and solving differential equations

K. E. Atkinson, *An Introduction to Numerical Analysis*, Wiley, 1978. Chapter 6 contains a discussion of the literature on the numerical solution of ordinary differential equations.

G. Birkhoff and G. Rota, *Ordinary Differential Equations*, 4th ed., Wiley, 1989.

N. B. Tufillaro, T. Abbott, and J. Reilly, *An Experimental Approach to Nonlinear Dynamics and Chaos*, Addison–Wesley, 1992. Appendix C discusses an earlier version of ode.

N. B. Tufillaro, E. F. Redish, and J. S. Risley, "`ode`: A numerical simulation of ordinary differential equations," pp. 480–481 in *Proceedings of the Conference on Computers in Physics Instruction*, Addison–Wesley, 1990.

9 libplot, a Function Library

9.1 Programming with libplot: An overview

GNU `libplot` is a free function library for drawing two-dimensional vector graphics. It can produce smooth, double-buffered animations for the X Window System, and can export files in many graphics file formats. It is 'device-independent' in the sense that its API (application programming interface) is to a large extent independent of the display type or output format.

There are bindings for C, C++, and other languages. The C binding, which is the most frequently used, is also called `libplot`, and the C++ binding, when it needs to be distinguished, is called `libplotter`. In this section we use `libplot` to refer to the library itself, irrespective of binding.

The graphical objects that `libplot` can draw include paths, circles and ellipses, points, markers, and 'adjusted labels' (justified text strings). A *path* is a sequence of line segments, circular arcs, elliptic arcs, quadratic Bezier curves, and/or cubic Bezier curves. Paths may be open or closed. User-specified filling of paths, circles, and ellipses is supported (fill rule and fill color, as well as pen color, may be specified). There is support for maintaining a Postscript-style stack of graphics contexts, i.e., a stack of drawing attribute sets. Path-related attributes include line thickness, line type, cap type, and join type, and text-related attributes include font name, font size, and text angle.

The fundamental abstraction provided by `libplot` is that of a *Plotter*. A Plotter is an object with an interface for the drawing of vector graphics which is similar to the interface provided by a traditional pen plotter. There are many types of Plotter, which differ in the output format they produce. Any number of Plotters, of the same or different types, may exist simultaneously in an application.

The drawing operations supported by Plotters of different types are identical, in agreement with the principle of device independence. So a graphics application that is linked with `libplot` may easily be written so as to produce output in any or all of the supported output formats.

The following are the currently supported types of Plotter.

- X Plotters. An X Plotter, when opened, pops up a window on an X Window System display and draws graphics in it. The window will be 'spun off' when the Plotter is closed; if it is subsequently reopened, a new window will be popped up. A spun-off window will remain on the screen but will vanish if you type '`q`' or click your mouse in it. Future releases may permit X Plotters, when reopened, to reuse an existing window.
- X Drawable Plotters. An X Drawable Plotter draws graphics in one or two specified drawables associated with an X Window System display. A 'drawable' is either a window or a pixmap. The drawables must be passed to the Plotter as parameters. (See Section 9.5 [Driver Parameters], page 118.)
- PNM Plotters. A PNM Plotter produces a single page of output in "portable anymap" format, and directs it to a file or other specified output stream. There are three types of portable anymap: PBM (portable bitmap, for monochrome graphics), PGM (portable

graymap), and PPM (portable pixmap, for colored graphics). The output file will be in whichever of these three formats is most appropriate. The file may be viewed or edited with many applications, such as the free image display application `xv`.

- GIF Plotters. A GIF Plotter produces a single page of output in a pseudo-GIF format. Unlike true GIF format, the pseudo-GIF format does not use LZW compression: it uses run-length encoding instead. So it does not transgress the Unisys patent that restricts the use of LZW compression. However, the output file may be viewed or edited with any application that understands GIF format, such as `xv`. The creation of animated pseudo-GIFs is supported.
- Illustrator Plotters. An Illustrator Plotter produces a single page of output in the format used by Adobe Illustrator, and directs it to a file or other specified output stream. The file may be edited with Adobe Illustrator (version 5, and more recent versions), or other applications.
- Postscript Plotters. A Postscript Plotter produces Postscript output and directs it to a file or other specified output stream. If only a single page of graphics is drawn on the Plotter then its output is in EPS (encapsulated Postscript) format, so it may be included in another document. It may also be edited with the `idraw` drawing editor.
- Fig Plotters. A Fig Plotter produces a single page of output in Fig format and directs it to a file or other specified output stream. The output may be edited with the `xfig` drawing editor. The `xfig` editor will export drawings in various other formats for inclusion in documents.
- PCL Plotters. A PCL Plotter produces output in PCL 5 format and directs it to a file or other specified output stream. PCL 5 is a powerful version of Hewlett–Packard's Printer Control Language, which supports vector graphics. The output may be sent to a PCL 5 device such as a LaserJet printer or high-end inkjet.
- HP-GL Plotters. An HP-GL Plotter produces output in the Hewlett–Packard Graphics Language (by default, in HP-GL/2), and directs it to a file or other specified output stream. The output may be imported into another application, or sent to a plotter.
- Tektronix Plotters. A Tektronix Plotter produces output in Tektronix 4014 format and directs it to a file or other specified output stream. The output may be displayed on any Tektronix 4014 emulator. Such an emulator is built into `xterm`, the X Window System terminal emulation program. The MS-DOS version of `kermit` also includes such an emulator.
- Metafile Plotters. A Metafile Plotter produces output in GNU graphics metafile format and directs it to a file or other specified output stream. This format is an extended version of the 'plot(5)' format found on some other operating systems. (See Appendix C [Metafiles], page 147.) It may be translated to other formats by an invocation of GNU `plot`. (See Chapter 3 [plot], page 27.)

A distinction among these types of Plotter is that all except X and X Drawable Plotters write graphics to a file or other output stream. An X Plotter pops up its own windows, and an X Drawable Plotter draws graphics in one or two X drawables.

Another distinction is that the first four types of Plotter (X, X Drawable, PNM, and GIF) produce bitmap output, while the remaining types produce output in a vector graphics format. In bitmap output the structure of the graphical objects is lost, but in a vector format it is retained.

An additional distinction is that X, X Drawable, Tektronix and Metafile Plotters are real-time. This means that they draw graphics or write to an output stream as the drawing operations are invoked on them. The remaining types of Plotter are not real-time, since their output streams can only be emitted after all functions have been called. For PNM and GIF Plotters, this is because the bitmap must be constructed before it is written out. For Illustrator and Postscript Plotters, it is because a 'bounding box' line must be placed at the head of the output file. For a Fig Plotter, it is because color definitions must be placed at the head of the output file.

The most important operations supported by any Plotter are `openpl` and `closepl`, which open and close it. Graphics may be drawn, and drawing attributes set, only within an `openpl`...`closepl` pair. The graphics produced within each `openpl`...`closepl` pair constitute a 'page'. In principle, any Plotter may be opened and closed arbitrarily many times. An X Plotter displays each page in a separate X window, and Postscript, PCL, and HP-GL Plotters render each page as a separate physical page. X Drawable Plotters and Tektronix Plotters manipulate a single drawable or display, on which pages are displayed in succession. Plotters that do not draw in real time (PNM, GIF, Illustrator, Postscript, Fig, PCL, and HP-GL Plotters) may wait until their existence comes to an end (i.e., until they are deleted) before outputting their pages of graphics.

In the current release of `libplot`, Postscript Plotters delay outputting graphics in this way, but PCL and HP-GL Plotters output each page of graphics individually, i.e., when `closepl` is invoked. PNM, GIF, Illustrator and Fig Plotters are similar, but output only the first page. That is because PNM, GIF, Illustrator and Fig formats support only a single page of graphics.

There are several other basic operations which any Plotter supports. The 'graphics display' drawn in by a Plotter is a square or rectangular region on a display device. But when using any Plotter to draw graphics, a user will specify the coordinates of graphical objects in device-independent 'user coordinates', rather than in device coordinates. A Plotter relates the user coordinate system to the device coordinate system by performing an affine transformation, which must be specified by the user.

Immediately after invoking `openpl` to open a Plotter, an application should invoke the `space` operation to initialize this transformation. This invocation specifies the rectangular region (in user coordinates) that will be mapped by the Plotter to the graphics display. The affine transformation may be updated at any later time by invoking `space` again, or by invoking `fconcat`. The `fconcat` operation will 'concatenate' (i.e., compose) the current affine transformation with any specified affine transformation. This sort of concatenation is a capability familiar from, e.g., Postscript.

Each Plotter maintains a Postscript-style stack of graphics contexts. This makes possible the rapid, efficient drawing of complicated pages of graphics. A graphics context includes the current affine transformation from the user coordinate system to the device coordinate system. It also includes such modal drawing attributes as graphics cursor position, linemode, line thickness, pen and fill colors, and the font used for drawing text. The state of any uncompleted path (if any) is included as well, since paths may be drawn incrementally, one portion (line segment or arc) at a time. The current graphics context is pushed onto the stack by calling `savestate`, and popped off by calling `restorestate`.

To permit vector graphics animation, any page of graphics may be split into 'frames'. A frame is ended, and a new frame is begun, by invoking the `erase` operation. On a Plotter that does real-time plotting (i.e., an X, X Drawable, Tektronix, or Metafile Plotter), this erases all plotted objects from the graphics display, allowing a new frame to be drawn. Displaying a sequence of frames in succession creates the illusion of smooth animation.

On most Plotters that do not do real-time plotting (i.e., PNM, Illustrator, Postscript, Fig, PCL, or HP-GL Plotters), invoking `erase` deletes all plotted objects from an internal buffer. For this reason, most Plotters that do not do real-time plotting will display only the final frame of any multiframe page.

GIF Plotters are in a class by themselves. Even though they do not do real time plotting, a GIF Plotter can produce multi-image output, i.e., an animated pseudo-GIF file, from a multiframe page. As noted above, the pseudo-GIF file produced by a GIF Plotter will contain only the first page of graphics. But if this page consists of multiple frames, then each invocation of `erase`, after the first, will be treated by default as a separator between successive images.

9.2 C Programming with `libplot`

9.2.1 The C application programming interface

`libplot` has bindings for several programming languages. Regardless of which language binding is used, the concepts behind `libplot` (Plotters, and a fixed set of operations that can be applied to any Plotter) remain the same. However, the ways in which Plotters are manipulated (created, selected for use, and deleted) may differ between bindings. This section discusses the C binding.

If you are writing an application in C that will use `libplot` to draw vector graphics, the first thing you need to know is how, in the C binding, Plotters are manipulated. There are four special functions for this, which you can invoke: `newpl`, `selectpl`, `deletepl`, and the parameter-setting function `parampl`. Actually, you would usually invoke them under the names `pl_newpl`, `pl_selectpl`, `pl_deletepl`, and `pl_parampl`. That is because in the C binding, the names of all functions begin with "pl_" unless the header file `plotcompat.h` is included. See Section 9.2.2 [C Compiling and Linking], page 88.

The `pl_newpl` function will create a Plotter of specified type. Its first argument may be "X", "Xdrawable", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", or "meta". It returns a nonnegative integer (a "handle") that may be used to refer to the Plotter. Before using a Plotter that you have created (i.e., before invoking any of the `libplot` operations on it), you must select the Plotter for use by calling `pl_selectpl`. Only one Plotter may be selected at a time, but by calling `pl_selectpl` you may switch from Plotter to Plotter at any time, even when the selected Plotter is open. A Plotter that is not currently selected can be deleted, and its storage freed, by calling `pl_deletepl`.

Strictly speaking, you do not need to call `pl_newpl`, `pl_selectpl`, or `pl_deletepl` in order to draw graphics. That is because at startup, a single Metafile Plotter that writes to standard output (with handle '0') is automatically created and selected. The presence of this default Plotter is for compatibility with pre-GNU versions of `libplot`. Of course, you may not wish to obtain output in metafile format, and you may not wish to write to standard output.

You should get into the habit of calling `pl_deletepl` whenever you are finished using a Plotter. In general, Plotters that do not plot graphics in real time (Postscript Plotters in particular) write out graphics only when the plot is finished, and `pl_deletepl` is called.

The following table summarizes the action of the four functions in the C binding that are used for Plotter manipulation.

int **`pl_newpl`** (const char **type*, FILE **infile*, FILE **outfile*, FILE **errfile*);

: `pl_newpl` creates a Plotter of type *type*, where *type* may be "X", "Xdrawable", "pnm", "gif", "ai", "ps", "fig", "pcl", "hpgl", "tek", or "meta". The Plotter will have input stream *infile*, output stream *outfile*, and error stream *errfile*. Any or all of these three may be NULL. Currently, all Plotters are write-only, so *infile* is ignored. X Plotters and X Drawable Plotters write graphics to an X Window System display rather than to an output stream, so if *type* is "X" or "Xdrawable" then *outfile* is ignored as well. Error messages (if any) are written to the stream *errfile*, unless *errfile* is NULL. The return value from `newpl` is a 'handle': a nonnegative integer by which the newly created Plotter is referred to. A negative return value indicates the Plotter could not be created.

int **`pl_selectpl`** (int *handle*);

: `pl_selectpl` selects a Plotter, referred to by its handle, for use. If any of the basic `libplot` operations is subsequently invoked, it will be applied to the selected Plotter. Only one Plotter may be selected at a time. At startup, a single Metafile Plotter that writes to standard output (with handle '0') is automatically created and selected.

: The handle of the previously selected Plotter is returned. A negative return value indicates the specified Plotter does not exist or could not be selected.

int **`pl_deletepl`** (int *handle*);

: `pl_deletepl` deletes a Plotter, referred to by its handle. The Plotter must not be selected at the time it is deleted. A negative return value indicates the Plotter could not be deleted.

int **`pl_parampl`** (const char **parameter*, void **value*);

: `pl_parampl` sets the value of the device driver parameter *parameter* to *value*. Device driver parameters specify Plotter options. The parameter values in effect at the time any Plotter is created are copied into it. For most parameters, *value* should be a `char *`, i.e., a string. Unrecognized parameters are ignored. For a list of the recognized parameters and their meaning, see Section 9.5 [Driver Parameters], page 118.

Up to now we have not discussed the fourth function, `pl_parampl`. Even though the Plotter interface is largely Plotter-independent, it is useful to be able to specify certain aspects of a Plotter's behavior at the time it is created. Plotter behavior is captured by a manageable number of parameters, which we call *device driver parameters*. A value for any parameter can be specified by calling `pl_parampl`. This function does not operate on any particular Plotter: it belongs to the C binding as a whole. The parameter values used by any Plotter are constant over the lifetime of the Plotter, and are those that were in effect when the Plotter was created.

Actually, a slightly more sophisticated rule applies. If at Plotter creation time a parameter is set, the value specified by the most recent call to `pl_parampl` will be the value used by the Plotter. If at Plotter creation time a parameter is *not* set, its default value will be used, unless the parameter is string-valued and there is an environment variable of the same name, in which case the value of that environment variable will be used. This rule increases run-time flexibility: an application programmer may allow non-critical driver parameters to be specified by the user via environment variables. Once set, a parameter may be unset by the programmer by calling `pl_parampl` with a value argument of NULL. This further increases flexibility.

9.2.2 C compiling and linking

The source code for a graphics application written in C, if it is to use `libplot`, must contain the lines

```
#include <stdio.h>
#include <plot.h>
```

The header file `plot.h` is distributed with `libplot`, and should have been installed on your system where your C compiler will find it. It contains prototypes for each of the functions in `libplot`, and some miscellaneous definitions.

In current releases of `libplot`, the names of all functions begin with the prefix "pl_". For example, the `openpl` operation is invoked on a Plotter by calling the function `pl_openpl`. If you wish, you may maintain backward compatibility by also including the header file `plotcompat.h`. This header file redefines `openpl` as `pl_openpl`, and similarly for the other `libplot` functions.

To link your application with `libplot`, you would use the appropriate '`-l`' option(s) on the command line when compiling it. You would use

```
-lplot -lXaw -lXmu -lXt -lXext -lX11 -lm
```

or, in recent releases of the X Window System,

```
-lplot -lXaw -lXmu -lXt -lSM -lICE -lXext -lX11 -lm
```

(Alternatively, you may need to use '`-lplot -lXm -lXt -lXext -lX11 -lm`', '`-lplot -lXm -lXt -lXext -lX11 -lm -lc -lgen`', or '`-lplot -lXm -lXt -lXext -lX11 -lm -lc -lPW`', on systems that provide Motif widgets instead of Athena widgets. In recent releases of the X Window System, you would insert '`-lSM -lICE`'. Recent releases of Motif require '`-lXp`' as well.)

On some platforms, the directories in which `libplot` or the other libraries are stored must be specified on the command line. For example, the options '`-lXaw -lXmu -lXt -lSM -lICE -lXext -lX11`', which specify X Window System libraries, may need to be preceded by an option like '`-L/usr/X11/lib`'.

On most systems `libplot` is installed as a DLL (dynamically linked library, or 'shared' library). This means that the linking with your application will take place at run time rather than compile time. The environment variable LD_LIBRARY_PATH lists the directories which will be searched for DLL's at run time. For your application to be executable, this environment variable should include the directory in which libplot is stored.

9.2.3 Sample drawings in C

The following is a sample application, written in C, that invokes libplot operations to draw vector graphics. It draws an intricate and beautiful path (Bill Gosper's "C" curve, discussed as Item #135 in *HAKMEM*, MIT Artificial Intelligence Laboratory Memo #239, 1972). As the numeric constant MAXORDER (here equal to 12) is increased, the path will take on the shape of a curly letter "C", which is the envelope of a myriad of epicyclic octagons.

```
#include <stdio.h>
#include <plot.h>
#define MAXORDER 12

void draw_c_curve (double dx, double dy, int order)
{
  if (order >= MAXORDER)
    pl_fcontrel (dx, dy);     /* continue path along (dx, dy) */
  else
    {
      draw_c_curve (0.5 * (dx - dy), 0.5 * (dx + dy), order + 1);
      draw_c_curve (0.5 * (dx + dy), 0.5 * (dy - dx), order + 1);
    }
}

int main ()
{
  int handle;

  /* set a Plotter parameter */
  pl_parampl ("PAGESIZE", "letter");

  /* create a Postscript Plotter that writes to standard output */
  if ((handle = pl_newpl ("ps", stdin, stdout, stderr)) < 0)
    {
      fprintf (stderr, "Couldn't create Plotter\n");
      return 1;
    }
  pl_selectpl (handle);       /* select the Plotter for use */

  if (pl_openpl () < 0)       /* open Plotter */
    {
      fprintf (stderr, "Couldn't open Plotter\n");
      return 1;
    }
  pl_fspace (0.0, 0.0, 1000.0, 1000.0); /* specify user coor system */
  pl_flinewidth (0.25);       /* line thickness in user coordinates */
  pl_pencolorname ("red");    /* path will be drawn in red */
```

```
    pl_erase ();                /* erase Plotter's graphics display */
    pl_fmove (600.0, 300.0);    /* position the graphics cursor */
    draw_c_curve (0.0, 400.0, 0);
    if (pl_closepl () < 0)      /* close Plotter */
      {
        fprintf (stderr, "Couldn't close Plotter\n");
        return 1;
      }
    pl_selectpl (0);            /* select default Plotter */
    if (pl_deletepl (handle) < 0) /* delete Plotter we used */
      {
        fprintf (stderr, "Couldn't delete Plotter\n");
        return 1;
      }
    return 0;
  }
```

As you can see, this application begins by calling the `pl_newpl` function to create a Postscript Plotter. The Postscript Plotter will produce output for a US letter-sized page, though any other standard page size, e.g., "a4", could be substituted. This would be arranged by altering the call to `pl_parampl`. The `PAGESIZE` parameter is one of several Plotter parameters that an application programmer may set by calling `pl_parampl`. For a complete list, see Section 9.5 [Driver Parameters], page 118.

After the Plotter is created, the application selects it for use, opens it, and draws the "C" curve recursively. The drawing of the curve is accomplished by calling the `pl_fmove` function to position the Plotter's graphics cursor, and then calling `draw_c_curve`. This subroutine repeatedly calls `pl_fcontrel`. The `pl_fcontrel` function continues a path by adding a line segment to it. The endpoint of each line segment is specified in relative coordinates, i.e., as an offset from the previous cursor position. After the "C" curve is drawn, the Plotter is closed. A Postscript file is written to standard output when `pl_deletepl` is called to delete the Plotter.

Specifying "pnm", "gif", "ai", "fig", "pcl", "hpgl", "tek", or "meta" as the first argument in the call to `pl_newpl`, instead of "ps", would yield a Plotter that would write graphics to standard output in the specified format, instead of Postscript. The `PAGESIZE` parameter is relevant to the "ai", "fig", "pcl", and "hpgl" output formats, but is ignored for the others. Specifying "meta" as the Plotter type may be useful if you wish to avoid recompilation for different output devices. Graphics metafile output may be piped to the `plot` utility and converted to any other supported output format, or displayed in an X window. See Chapter 3 [plot], page 27.

If "X" were specified as the first argument of `pl_newpl`, the curve would be drawn in a popped-up X window, and the output stream argument would be ignored. Which X Window System display the window would pop up on would be determined by the `DISPLAY` parameter, or if that parameter were not set, by the `DISPLAY` environment variable. The size of the X window would be determined by the `BITMAPSIZE` parameter, or if that parameter were not set, by the `BITMAPSIZE` environment variable. The default value is "570x570". For the "pnm" and "gif" Plotter types, the interpretation of `BITMAPSIZE` is similar.

You could also specify "Xdrawable" as the Plotter type. For you to make this work, you would need to know a bit about X Window System programming. You would need to create at least one X drawable (i.e., window or a pixmap), and by invoking the `pl_parampl` function before `newpl` is called, set it as the value of the parameter `XDRAWABLE_DRAWABLE1` or `XDRAWABLE_DRAWABLE2`. For the parameters that affect X Drawable Plotters, see Section 9.5 [Driver Parameters], page 118.

The following is another sample application, written in C, that invokes `libplot` operations to draw vector graphics. It draws a spiral consisting of elliptically boxed text strings, each of which reads "GNU libplot!". This figure will be sent to standard output in Postscript format.

```
#include <stdio.h>
#include <plot.h>
#include <math.h>
#define SIZE 100.0   /* nominal size of user coordinate frame */
#define EXPAND 2.2   /* expansion factor for elliptical box */

void draw_boxed_string (char *s, double size, double angle)
{
  double true_size, width;

  pl_ftextangle (angle);         /* text inclination angle (degrees) */
  true_size = pl_ffontsize (size);  /* choose font size */
  width = pl_flabelwidth (s);    /* compute width of text string */
  pl_fellipserel (0.0, 0.0,      /* draw surrounding ellipse */
             EXPAND * 0.5 * width, EXPAND * 0.5 * true_size, angle);
  pl_alabel ('c', 'c', s);       /* draw centered text string */
}

int main()
{
  int handle, i;

  /* set a Plotter parameter */
  pl_parampl ("PAGESIZE", "letter");

  /* create a Postscript Plotter that writes to standard output */
  if ((handle = pl_newpl ("ps", stdin, stdout, stderr)) < 0)
    {
      fprintf (stderr, "Couldn't create Plotter\n");
      return 1;
    }
  pl_selectpl (handle);         /* select the Plotter for use */

  if (pl_openpl () < 0)         /* open Plotter */
    {
      fprintf (stderr, "Couldn't open Plotter\n");
      return 1;
    }
```

```
  pl_fspace (-(SIZE), -(SIZE), SIZE, SIZE); /* spec. user coor system */
  pl_pencolorname ("blue");     /* pen color will be blue */
  pl_fillcolorname ("white");
  pl_filltype (1);              /* ellipses will be filled with white */
  pl_fontname ("NewCenturySchlbk-Roman"); /* choose a Postscript font */

  for (i = 80; i > 1; i--)      /* loop through angles */
    {
      double theta, radius;

      theta = 0.5 * (double)i;  /* theta is in radians */
      radius = SIZE / pow (theta, 0.35);  /* this yields a spiral */
      pl_fmove (radius * cos (theta), radius * sin (theta));
      draw_boxed_string ("GNU libplot!", 0.04 * radius,
                         (180.0 * theta / M_PI) - 90.0);
    }

  if (pl_closepl () < 0)        /* close Plotter */
    {
      fprintf (stderr, "Couldn't close Plotter\n");
      return 1;
    }
  pl_selectpl (0);
  if (pl_deletepl (handle) < 0) /* delete Plotter we used */
    {
      fprintf (stderr, "Couldn't delete Plotter\n");
      return 1;
    }
  return 0;
}
```

This example shows what is involved in plotting a text string or text strings. First, the desired font must be retrieved. A font is fully specified by calling `pl_fontname`, `pl_fontsize`, and `pl_textangle`, or their floating point counterparts `pl_ffontname`, `pl_ffontsize`, and `pl_ftextangle`. Since these three functions may be called in any order, each of them returns the size of the font that it selects, as a convenience to the programmer. This may differ slightly from the size specified in the most recent call to `pl_fontsize` or `pl_ffontsize`, since many Plotters have only a limited repertory of fonts. The above example plots each text string in the "NewCenturySchlbk-Roman" font, which is available on Postscript Plotters. See Section A.1 [Text Fonts], page 124.

If you replace "ps" by "X" in the call to `pl_newpl`, an X Plotter rather than a Postscript Plotter will be used, and the spiral will be drawn in a popped-up X window. If your X display does not support the "NewCenturySchlbk-Roman" font, you may substitute any other scalable font, such as the widely available "utopia-medium-r-normal". For the format of font names, see Section A.3 [Text Fonts in X], page 131. If the X Plotter is unable to retrieve the font you specify, it will first attempt to use a default scalable font ("Helvetica"), and if that fails, use a default Hershey vector font ("HersheySerif") instead. Hershey fonts are constructed from line segments, so each built-in Hershey font is available on all types of Plotter.

If you are using an older (pre-X11R6) X Window System display, you will find that retrieving a scalable font is a time-consuming operation. The above example may run slowly on some older X displays, since a new font must be retrieved before each text string is drawn. That is because each text string has a different angle of inclination. It is possible to retrieve individual characters from an X11R6 display, rather than retrieving an entire rasterized font. If this feature is available, the X Plotter will automatically take advantage of it to save time.

9.2.4 Animated GIFs in C

Using `libplot` to create pseudo-GIF files, including animated pseudo-GIFs, is straightforward. A GIF Plotter is a Plotter like any other, and it supports the same drawing operations. However, it has two special properties. (1) It can draw only a single page of graphics, i.e., only the graphics contained in the first `openpl`...`closepl` appear in the output file. In this, it resembles other Plotters that do not plot in real time. (2) Within this page, each invocation of `erase` is normally treated as the beginning of a new image in the output file. There is an exception to this: the first invocation of `erase` begins a new image only if something has already been drawn.

The reason for the exception is that many programmers who use `libplot` are in the habit of invoking `erase` immediately after a Plotter is opened. This is not a bad habit, since a few types of Plotter (e.g., X Drawable and Tektronix Plotters) are 'persistent' in the sense that previously drawn graphics remain visible.

The following program creates a simple animated pseudo-GIF, 150 pixels wide and 100 pixels high.

```
#include <stdio.h>
#include <plot.h>

int main()
{
  int i, handle;

  /* set Plotter parameters */
  pl_parampl ("BITMAPSIZE", "150x100");
  pl_parampl ("BG_COLOR", "orange");
  pl_parampl ("TRANSPARENT_COLOR", "orange");
  pl_parampl ("GIF_ITERATIONS", "100");
  pl_parampl ("GIF_DELAY", "5");

  /* create a GIF Plotter with the specified parameters */
  handle = pl_newpl ("gif", stdin, stdout, stderr);
  pl_selectpl (handle);         /* select the Plotter for use */

  pl_openpl(); /* begin page of graphics */
  pl_space (0, 0, 149, 99); /* specify user coordinate system */

  pl_pencolorname ("red");      /* objects will be drawn in red */
  pl_linewidth (5);             /* set the line thickness */
  pl_filltype (1);              /* objects will be filled */
```

```
    pl_fillcolorname ("black");   /* set the fill color */

    for (i = 0; i < 180 ; i += 15)
      {
        pl_erase ();              /* begin new GIF image */
        pl_ellipse (75, 50, 40, 20, i); /* draw an ellipse */
      }
    pl_closepl ();                /* end page of graphics */
    pl_selectpl (0);              /* select default Plotter */
    pl_deletepl (handle);         /* delete Plotter we used */
    return 0;
  }
```

The animated pseudo-GIF will be written to standard output. It will consist of twelve images, showing the counterclockwise rotation of a black-filled red ellipse through 180 degrees. The pseudo-GIF will be 'looped' (see below), so the ellipse will rotate repeatedly.

The parameters of the ellipse are expressed in terms of user coordinates, not pixel coordinates. But the call to `pl_space` defines user and pixel coordinates to be effectively the same. User coordinates are chosen so that the lower left corner is (0,0) and the upper right corner is (149,99). Since this agrees with the image size, individual pixels may be addressed in terms of integer user coordinates. For example, `pl_point(149,99)` would set the pixel in the upper right hand corner of the image to the current pen color.

Besides `BITMAPSIZE` and `BG_COLOR`, there are several important GIF Plotter parameters that may be set with the `pl_parampl` function. The `TRANSPARENT_COLOR` parameter may be set to the name of a color. Pixels in a pseudo-GIF that have that color will be treated as transparent by most software. This is usually used to create a transparent background. In the example above, the background color is specified as orange, but the transparent color is also specified as orange. So the background will not actually be displayed.

The `GIF_ITERATIONS` parameter, if set, specifies the number of times that a multi-frame pseudo-GIF should be looped. The `GIF_DELAY` parameter specifies the number of hundredths of a seconds that should elapse between successive images.

The `INTERLACE` parameter is sometimes useful. If it is set to `"yes"`, the pseudo-GIF will be interlaced. This is of greatest value for single-frame GIFs. For full details on Plotter parameters, see Section 9.5 [Driver Parameters], page 118.

9.2.5 X Window System animations in C

You may use `libplot` to produce vector graphics animations on any Plotter that does real-time plotting (i.e., an X, X Drawable, Tektronix, or Metafile Plotter). By definition, the 'frames' in any page of graphics are separated by invocations of `erase`. So the graphics display will be cleared after each frame. If successive frames differ only slightly, a smooth animation will result.

The following is a sample application, written in C, that produces an animation for the X Window System. It displays a 'drifting eye'. As the eye drifts across a popped-up window from left to right, it slowly rotates. After the eye has drifted across twice, the window will vanish.

```
#include <stdio.h>
#include <plot.h>

int main ()
{
  int handle, i = 0, j;

  /* set Plotter parameters */
  pl_parampl ("BITMAPSIZE", "300x150");
  pl_parampl ("VANISH_ON_DELETE", "yes");
  pl_parampl ("USE_DOUBLE_BUFFERING", "yes");

  /* create an X Plotter with the specified parameters */
  if ((handle = pl_newpl ("X", stdin, stdout, stderr)) < 0)
    {
      fprintf (stderr, "Couldn't create Plotter\n");
      return 1;
    }
  pl_selectpl (handle);       /* select the Plotter for use */

  if (pl_openpl () < 0)       /* open Plotter */
    {
      fprintf (stderr, "Couldn't open Plotter\n");
      return 1;
    }
  pl_space (0, 0, 299, 149);  /* specify user coordinate system */
  pl_linewidth (8);           /* line thickness in user coordinates */
  pl_filltype (1);            /* objects will be filled */
  pl_bgcolorname ("saddle brown"); /* background color for the window*/
  for (j = 0; j < 300; j++)
    {
      pl_erase ();            /* erase window */
      pl_pencolorname ("red"); /* choose red pen, with cyan filling */
      pl_fillcolorname ("cyan");
      pl_ellipse (i, 75, 35, 50, i);  /* draw an ellipse */
      pl_colorname ("black"); /* choose black pen, with black filling */
      pl_circle (i, 75, 12);  /* draw a circle [the pupil] */
      i = (i + 2) % 300;      /* shift rightwards */
    }
  if (pl_closepl () < 0)         /* close Plotter */
    {
      fprintf (stderr, "Couldn't close Plotter\n");
      return 1;
    }
```

```
    pl_selectpl (0);             /* select default Plotter */
    if (pl_deletepl (handle) < 0) /* delete Plotter we used */
      {
        fprintf (stderr, "Couldn't delete Plotter\n");
        return 1;
      }
    return 0;
  }
```

As you can see, this application begins by calling `pl_parampl` several times to set device driver parameters, and then calls `pl_newpl` to create an X Plotter. The X Plotter window will have size 300x150 pixels. This window will vanish when the Plotter is deleted. If the `VANISH_ON_DELETE` parameter were not set to "yes", the window would remain on the screen until removed by the user (by typing '`q`' in it, or by clicking with a mouse).

Setting the parameter `USE_DOUBLE_BUFFERING` to "yes" requests that double buffering be used. This is very important if you wish to produce a smooth animation, with no jerkiness. Normally, an X Plotter draws graphics into a window in real time, and erases the window when `pl_erase` is called. But if double buffering is used, each frame of graphics is written into an off-screen buffer, and is copied into the window, pixel by pixel, when `pl_erase` is called or the Plotter is closed. This is a bit counterintuitive, but is exactly what is needed for smooth animation.

After the Plotter is created, it is selected for use and opened. When `pl_openpl` is called, the window pops up, and the animation begins. In the body of the `for` loop there is a call to `pl_erase`, and also a sequence of `libplot` operations that draws the eye. The pen color and fill color are changed twice with each passage through the loop. You may wish to experiment with the animation parameters to produce the best effects on your video hardware.

The locations of the objects that are plotted in the animation are expressed in terms of user coordinates, not pixel coordinates. But the call to `pl_space` defines user and pixel coordinates to be effectively the same. User coordinates are chosen so that the lower left corner is (0,0) and the upper right corner is (299,149). Since this agrees with the window size, individual pixels may be addressed in terms of integer user coordinates. For example, `pl_point(299,149)` would set the pixel in the upper right hand corner of the window to the current pen color.

The following is another sample animation, this time of a rotating letter 'A'.

```
#include <stdio.h>
#include <plot.h>

int main()
{
  int handle, angle = 0;

  /* set Plotter parameters */
  pl_parampl ("BITMAPSIZE", "300x300");
  pl_parampl ("BG_COLOR", "blue"); /* background color for window */
  pl_parampl ("USE_DOUBLE_BUFFERING", "yes");
```

```
    /* create an X Plotter with the specified parameters */
    handle = pl_newpl ("X", stdin, stdout, stderr);
    pl_selectpl (handle);

    /* open X Plotter, initialize coordinates, pen, and font */
    pl_openpl ();
    pl_fspace (0.0, 0.0, 1.0, 1.0);  /* use normalized coordinates */
    pl_pencolorname ("white");
    pl_ffontsize (1.0);
    pl_fontname ("NewCenturySchlbk-Roman");

    pl_fmove (.50,.50);             /* move to center */
    while (1)                       /* loop endlessly */
      {
        pl_erase ();
        pl_textangle (angle++);      /* set new rotation angle */
        pl_alabel ('c', 'c', "A");   /* draw a centered 'A' */
      }
    pl_closepl();                   /* close Plotter */

    pl_selectpl (0);                /* select default Plotter */
    pl_deletepl (handle);           /* delete Plotter we used */
    return 0;
  }
```

This animation serves as a good test of the capabilities of an X Window System display. On a modern X11R6 display, animation will be smooth and fast. That is because X11R6 displays can rasterize individual characters from a font without rasterizing the entire font. If your X display does not support the "NewCenturySchlbk-Roman" font, you may substitute any other scalable font, such as the widely available "utopia-medium-r-normal". For the format of font names, see Section A.3 [Text Fonts in X], page 131. If the X Plotter is unable to retrieve the font you specify, it will first attempt to use a default scalable font ("Helvetica"). If that too fails, it will use a default Hershey vector font ("HersheySerif") instead.

Animations that use Hershey fonts are normally faster than ones that use Postscript fonts or other X Window System fonts, since the Hershey fonts are constructed from line segments. Rasterizing line segments can be done rapidly. But if you use a scalable font such as "NewCenturySchlbk-Roman" or "utopia-medium-r-normal", you will notice that the rotation speeds up after the letter 'A' has rotated through 360 degrees. That is because the 'A' at angles past 360 degrees has already been rasterized.

9.2.6 Advanced X Window System programming

Applications that run under the X Window System are normally built using Xt, the X Toolkit. In Xt, an application is constructed from 'widgets' such as text entry fields, buttons, sliders, drawing areas, etc. When the application starts up, each widget is configured to respond appropriately to 'events', which include key presses and mouse clicks. After the widgets are configured, control is transferred to the Xt event loop.

GNU libplot can be used within the Xt event loop to draw vector graphics. For this, it would use one or more X Drawable Plotters. An X Drawable Plotter is a Plotter that can plot into an off-screen pixmap or an on-screen window, such as a window associated with a widget.

The following sample application shows how an X Drawable Plotter would be used. The application draws a 'C' curve, as defined in a previous section, in a popped-up window. The usual Xt command-line options may be used: the window background color is specified with the '-bg' option, the window geometry with '-geometry', etc. The curve is initially drawn in red, but clicking once with the mouse will redraw it in green. A second mouse click will redraw it in red, and so forth. The application will terminate when 'q' is typed.

```
#include <stdio.h>
#include <plot.h>
#include <X11/Xlib.h>
#include <X11/Intrinsic.h>
#include <X11/Shell.h>
#include <X11/StringDefs.h>
#include <X11/Core.h>
int green = 0;                /* draw in green, not red? */

#define MAXORDER 12
void draw_c_curve (double dx, double dy, int order)
{
  if (order >= MAXORDER)
    pl_fcontrel (dx, dy);      /* continue path along (dx, dy) */
  else
    {
      draw_c_curve (0.5 * (dx - dy), 0.5 * (dx + dy), order + 1);
      draw_c_curve (0.5 * (dx + dy), 0.5 * (dy - dx), order + 1);
    }
}
void Redraw (Widget w, XEvent *ev, String *params, Cardinal *n_params)
{
  /* draw C curve */
  pl_erase ();
  pl_pencolorname (green ? "green" : "red");
  pl_fmove (600.0, 300.0);
  draw_c_curve (0.0, 400.0, 0);
  pl_endpath ();
}

void Toggle (Widget w, XEvent *ev, String *params, Cardinal *n_params)
{
  green = (green ? 0 : 1);
  Redraw (w, ev, params, n_params);
}
```

```
void Quit (Widget w, XEvent *ev, String *params, Cardinal *n_params)
{
  exit (0);
}

/* mapping of events to actions */
static const String translations =
"<Expose>:      redraw()\n\
<Btn1Down>:     toggle()\n\
<Key>q:         quit()";

/* mapping of actions to subroutines */
static XtActionsRec actions[] =
{
  {"redraw",            Redraw},
  {"toggle",            Toggle},
  {"quit",              Quit},
};

/* default parameters for widgets */
static String default_resources[] =
{
  "Example*geometry:      250x250",
  (String)NULL
};

int main (int argc, char *argv[])
{
  Arg wargs[10];                /* storage of widget args */
  Display *display;             /* X display */
  Widget shell, canvas;         /* toplevel widget; child */
  Window window;                /* child widget's window */
  XtAppContext app_con;         /* application context */
  int handle, i;
  char *bg_colorname = "white";

  /* take background color from command line */
  for (i = 0; i < argc - 1; i++)
    if (strcmp (argv[i], "-bg") == 0)
      bg_colorname = argv[i + 1];

  /* create toplevel shell widget */
  shell = XtAppInitialize (&app_con,
                           (String)"Example", /* app class */
                           NULL,              /* options */
                           (Cardinal)0,       /* num of options */
                           &argc,             /* command line */
                           argv,              /* command line */
                           default_resources,
                           NULL,              /* ArgList */
                           (Cardinal)0        /* num of Args */
                           );
```

```
    /* set default widget parameters (including window size) */
    XtAppSetFallbackResources (app_con, default_resources);

    /* map actions to subroutines */
    XtAppAddActions (app_con, actions, XtNumber (actions));

    /* create canvas widget as child of shell widget; realize both */
    XtSetArg(wargs[0], XtNargc, argc);
    XtSetArg(wargs[1], XtNargv, argv);
    canvas = XtCreateManagedWidget ((String)"", coreWidgetClass,
                                    shell, wargs, (Cardinal)2);
    XtRealizeWidget (shell);

    /* for the canvas widget, map events to actions */
    XtSetArg (wargs[0], XtNtranslations,
              XtParseTranslationTable (translations));
    XtSetValues (canvas, wargs, (Cardinal)1);

    /* initialize GNU libplot */
    display = XtDisplay (canvas);
    pl_parampl ("XDRAWABLE_DISPLAY", display);
    window = XtWindow (canvas);
    pl_parampl ("XDRAWABLE_DRAWABLE1", &window);
    pl_parampl ("BG_COLOR", bg_colorname);
    handle = pl_newpl ("Xdrawable", NULL, NULL, stderr);
    pl_selectpl (handle);
    pl_openpl ();
    pl_fspace (0.0, 0.0, 1000.0, 1000.0);
    pl_flinewidth (0.25);

    /* transfer control to X Toolkit event loop (doesn't return) */
    XtAppMainLoop (app_con);

    return 1;
  }
```

Even if you are not familiar with X Window System programming, the structure of this application should be clear. It defines three callbacks: `Redraw`, `Toggle`, and `Quit`. They are invoked respectively in response to (1) a window expose event or mouse click, (2) a mouse click, and (3) a typed '`q`'. The first drawing of the 'C' curve (in red) takes place because the window receives an initial expose event.

This example could be extended to take window resizing into account. Actually, X Drawable Plotters are usually used to draw vector graphics in off-screen pixmaps rather than on-screen windows. Pixmaps, unlike windows, are never resized.

9.3 C++ Programming with `libplotter`

9.3.1 The `Plotter` class

The C++ binding for `libplot` is provided by a class library, named `libplotter`. This library implements a `Plotter` class, of which all Plotters are instances. Actually, a Plotter would normally be an instance of an appropriate derived class, determined by the Plotter's output format. Derived classes currently include `XPlotter`, `XDrawablePlotter`, `PNMPlotter`, `GIFPlotter`, `AIPlotter`, `PSPlotter`, `FigPlotter`, `PCLPlotter`, `HPGLPlotter`, `TekPlotter`, and `MetaPlotter`. The names should be self-explanatory. The operations that may be applied to any Plotter (e.g., the `openpl` operation, which begins a page of graphics) are implemented as public function members of the `Plotter` class.

At the time a Plotter is created, its input, output, and error streams must be specified. (The first is ignored, since at present, all Plotters are write-only.) The streams may be specified either as iostreams or as `FILE` pointers. That is, the two constructors

```
Plotter(istream& instream, ostream& outstream, ostream& errstream);
Plotter(FILE *infile, FILE *outfile, FILE *errfile);
```

are provided for the base Plotter class, and similarly for each of its derived classes. So, for example, both

```
PSPlotter plotter(cin, cout, cerr);
```

and

```
PSPlotter plotter(stdin, stdout, stderr);
```

are possible declarations of a Postscript Plotter that writes to standard output. In the iostream case, an ostream with a null stream buffer may be specified as the output stream and/or the error stream, to request that no output take place. In the `FILE` pointer case, specifying a null `FILE` pointer would accomplish the same thing. Instances of the `XPlotter` and `XDrawablePlotter` classes always ignore the output stream argument, since they write graphics to an X Display rather than to a stream.

The parameter-setting function `parampl`, which plays an important role in all bindings of `libplot`, is implemented in `libplotter` as a static function member of the `Plotter` class. The following is a formal description.

int `Plotter::parampl` (const char **parameter*, void **value*);

> `Plotter::parampl` sets the value of the device driver parameter *parameter* to *value*. Device driver parameters specify Plotter options. The parameter values in effect at the time any Plotter is created are copied into it. For most parameters, *value* should be a `char *`, i.e., a string. Unrecognized parameters are ignored. For a list of the recognized parameters and their meaning, see Section 9.5 [Driver Parameters], page 118.

Just as in the C binding, `Plotter::parampl` gives the programmer fine control over the parameters of subsequently created Plotters. The parameter values used by any Plotter are constant over the lifetime of the Plotter, and are those that were set when the Plotter was created. If at Plotter creation time a parameter is *not* set, its default value will be used, unless the parameter is string-valued and there is an environment variable of the same name, in which case the value of that environment variable will be used.

Once set, a parameter may be unset by the programmer by calling `Plotter::parampl` with a value argument of NULL. This further increases flexibility.

9.3.2 C++ compiling and linking

The source code for a graphics application written in C++, if it is to use `libplotter`, must contain the line

```
#include <plotter.h>
```

The header file `plotter.h` is distributed with `libplotter`, and should have been installed on your system where your C++ compiler will find it. It declares the `Plotter` class and its derived class, and also contains some miscellaneous definitions. It includes the header files `<iostream.h>` and `<stdio.h>`, so you do not need to include them separately.

To link your application with `libplotter`, you would use the appropriate '`-l`' option(s) on the command line when compiling it. You would use

```
-lplotter -lXaw -lXmu -lXt -lXext -lX11 -lm
```

or, in recent releases of the X Window System,

```
-lplotter -lXaw -lXmu -lXt -lSM -lICE -lXext -lX11 -lm
```

(Alternatively, you may need to use '`-lplotter -lXm -lXt -lXext -lX11 -lm`', '`-lplotter -lXm -lXt -lXext -lX11 -lm -lc -lgen`', or '`-lplotter -lXm -lXt -lXext -lX11 -lm -lc -lPW`', on systems that provide Motif widgets instead of Athena widgets. In recent releases of the X Window System, you would insert '`-lSM -lICE`'. Recent releases of Motif require '`-lXp`' as well.)

On some platforms, the directories in which `libplotter` or the other libraries are stored must be specified on the command line. For example, the options '`-lXaw -lXmu -lXt -lSM -lICE -lXext -lX11`', which specify X Window System libraries, may need to be preceded by an option like '`-L/usr/X11/lib`'.

On most systems `libplotter` is installed as a DLL (dynamically linked library, or 'shared' library). This means that the linking with your application will take place at run time rather than compile time. The environment variable `LD_LIBRARY_PATH` lists the directories which will be searched for DLL's at run time. For your application to be executable, this environment variable should include the directory in which `libplotter` is stored.

9.3.3 Sample drawings in C++

In a previous section, there are several sample C programs that show how to draw vector graphics using `libplot`'s C binding. See Section 9.2.3 [Sample C Drawings], page 89. In this section, we give a modified version of one of the C programs, showing how `libplot`'s C++ binding, i.e., `libplotter`, can be used similarly.

The following C++ program draws an intricate and beautiful path (Bill Gosper's "C" curve).

```
#include <plotter.h>
#define MAXORDER 12

void draw_c_curve (Plotter& plotter, double dx, double dy, int order)
{
  if (order >= MAXORDER)
    plotter.fcontrel (dx, dy); // continue path along (dx, dy)
```

```
    else
      {
        draw_c_curve (plotter,
                      0.5 * (dx - dy), 0.5 * (dx + dy), order + 1);
        draw_c_curve (plotter,
                      0.5 * (dx + dy), 0.5 * (dy - dx), order + 1);
      }
}

int main ()
{
  // set a Plotter parameter
  Plotter::parampl ("PAGESIZE", "letter");

  PSPlotter plotter(cin, cout, cerr); // declare Plotter
  if (plotter.openpl () < 0)          // open Plotter
    {
      cerr << "Couldn't open Plotter\n";
      return 1;
    }

  plotter.fspace (0.0, 0.0, 1000.0, 1000.0); // specify user coor system
  plotter.flinewidth (0.25);       // line thickness in user coordinates
  plotter.pencolorname ("red");    // path will be drawn in red
  plotter.erase ();                // erase Plotter's graphics display
  plotter.fmove (600.0, 300.0);    // position the graphics cursor
  draw_c_curve (plotter, 0.0, 400.0, 0);
  if (plotter.closepl () < 0)      // close Plotter
    {
      cerr << "Couldn't close Plotter\n";
      return 1;
    }
  return 0;
}
```

The above is a straightforward translation of the corresponding C program. Here, `plotter` is declared as an instance of the `PSPlotter` class, which will write Postscript graphics to the output stream `cout`. The graphics are drawn by invoking member functions.

This object-oriented approach to drawing graphics is probably more natural than the approach used by the C binding. With `libplotter`, the use of multiple Plotters in an application becomes very easy. No switching between Plotters is required: the graphics operations can be immediately invoked on any existing Plotter.

9.4 The functions in `libplot`: A detailed listing

In the current release of GNU `libplot`, any Plotter supports 92 distinct operations. A language binding for `libplot` necessarily includes 92 functions that correspond to these operations. In the C binding, these 92 functions belong to the C API (application programming interface). In the C++ binding, they are implemented as public member functions of the `Plotter` class.

A language binding may also include functions for creating, selecting, and deleting Plotters. For example, the C binding includes the additional functions `pl_newpl`, `pl_selectpl`, and `pl_deletepl`. See Section 9.2.1 [The C API], page 86. In the C binding, the names of all functions should be preceded by "pl_" unless the header file `plotcompat.h` is included. See Section 9.2.2 [C Compiling and Linking], page 88.

The 92 functions that operate on a specified Plotter are divided into the four sets tabulated below.

1. Setup functions: functions that open, initialize, or close the Plotter.
2. Functions that cause the Plotter to draw objects.
3. Functions that set or affect the Plotter's drawing attributes.
4. Functions affecting the affine map used by the Plotter to transform user coordinates to device coordinates.

Many functions come in two versions: integer and double precision floating point. Internally, `libplot` uses double precision floating point. The integer versions are provided for backward compatibility. If there are two versions of a function, the name of the floating point version begins with the letter '`f`'.

Many functions come in both absolute and relative versions, also. The latter use relative coordinates (i.e., coordinates relative to the current position of the graphics cursor), and their names end in '`rel`'.

Currently, only a few of the 92 functions have meaningful return values.

9.4.1 Setup functions

The following are the "setup functions" in `libplot`. They are the basic functions that open, initialize, or close an already-created Plotter. They are listed in the approximate order in which they would be called.

In the C binding, the names of all functions should be preceded by "pl_", unless the header file `plotcompat.h` is included. See Section 9.2.2 [C Compiling and Linking], page 88. In the C++ binding, these are member functions of the `Plotter` class and its subclasses.

int `openpl` ();

> `openpl` opens a Plotter, i.e., begins a page of graphics. This resets the Plotter's drawing attributes to their default values. A negative return value indicates the Plotter could not be opened.
>
> Currently, an X Plotter pops up a new window on an X Window System display for each page of graphics, i.e., with each invocation of `openpl`. Future releases may support window re-use.

int `bgcolor` (int *red*, int *green*, int *blue*);

> `bgcolor` sets the background color for the Plotter's graphics display, using a 48-bit RGB color model. The arguments *red*, *green* and *blue* specify the red, green and blue intensities of the background color. Each is an integer in the

range 0x0000...0xffff, i.e., 0...65535. The choice (0, 0, 0) signifies black, and the choice (65535, 65535, 65535) signifies white.

bgcolor affects only Plotters that produce bitmaps, i.e., X Plotters, X Drawable Plotters, PNM Plotters, and GIF Plotters. Its effect is simple: the next time the erase operation is invoked on such a Plotter, its display will be filled with the specified color.

int bgcolorname (const char **name*);

bgcolorname sets the background color for the the graphics display to be *name*. For information on what color names are recognized, see Appendix B [Color Names], page 146. Unrecognized colors are interpreted as "white".

bgcolorname affects only Plotters that produce bitmaps, i.e., X Plotters, X Drawable Plotters, PNM Plotters, and GIF Plotters. Its effect is simple: the next time the erase operation is invoked on such a Plotter, its display will be filled with the specified color.

int erase ();

erase begins the next frame of a multiframe page, by clearing all previously plotted objects from the graphics display, and filling it with the background color (if any).

It is frequently useful to invoke erase at the beginning of each page, i.e., immediately after invoking openpl. That is because some Plotters are persistent, in the sense that objects drawn within an openpl...closepl pair remain on the graphics display even after a new page is begun by a subsequent invocation of openpl. Currently, only X Drawable Plotters and Tektronix Plotters are persistent. Future releases may support optional persistence for X Plotters also.

On X Plotters and X Drawable Plotters the effects of invoking erase will be altogether different if the device driver parameter USE_DOUBLE_BUFFERING is set to "yes". In this case, objects will be written to an off-screen buffer rather than to the graphics display, and invoking erase will (1) copy the contents of this buffer to the display, and (2) erase the buffer by filling it with the background color. This 'double buffering' feature facilitates smooth animation. See Section 9.5 [Driver Parameters], page 118.

int space (int *x0*, int *y0*, int *x1*, int *y1*);
int fspace (double *x0*, double *y0*, double *x1*, double *y1*);

space and fspace take two pairs of arguments, specifying the positions of the lower left corner and upper right corner of the graphics display, in user coordinates. In other words, calling space or fspace sets the affine transformation from user coordinates to device coordinates. One of these operations must be performed at the beginning of each page of graphics, i.e., immediately after openpl is invoked.

int space2 (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*);
int fspace2 (double *x0*, double *y0*, double *x1*, double *y1*, double *x2*, double *y2*);

space2 and fspace2 are extended versions of space and fspace, and may be used instead. Their arguments are the three defining vertices of an 'affine window' (a drawing parallelogram), in user coordinates. The specified vertices are the lower left, the lower right, and the upper left. This window will be mapped affinely onto the graphics display.

int **havecap** (const char *s);

> `havecap` tests whether or not a Plotter, which need not be open, has a specified capability. The return value is 0, 1, or 2, signifying no/yes/maybe. For unrecognized capabilities the return value is zero. Recognized capabilities include "WIDE_LINES" (i.e., the ability to draw lines with a non-default thickness), "DASH_ARRAY" (the ability to draw in arbitrary dashing styles, as requested by the `linedash` function), "SETTABLE_BACKGROUND" (the ability to set the color of the background), and "SOLID_FILL". The "HERSHEY_FONTS", "PS_FONTS", "PCL_FONTS", and "STICK_FONTS" capabilities indicate whether or not fonts of a particular class are supported. See Section A.1 [Text Fonts], page 124.
>
> All Plotters except Tektronix Plotters have the "SOLID_FILL" capability, meaning they can fill paths with solid color. Each such Plotter has at least one of the "EVEN_ODD_FILL" and "NONZERO_WINDING_NUMBER_FILL" capabilities. These indicate the supported rules for determining the 'inside' of a path.
>
> The 'maybe' value is returned for most capabilities by Metafile Plotters, which do no drawing themselves. The output of a Metafile Plotter must be translated to another format, or displayed, by invoking `plot`.

int **flushpl** ();

> `flushpl` flushes (i.e., pushes onward) all plotting commands to the display device. This is useful only if the currently selected Plotter does real-time plotting, since it may be used to ensure that all previously plotted objects have been sent to the display and are visible to the user. It has no effect on Plotters that do not do real-time plotting.

int **closepl** ();

> `closepl` closes a Plotter, i.e., ends a page of graphics. A negative return value indicates the Plotter could not be closed.
>
> In the present release of `libplot`, some Plotters output pages in real time, i.e., with each invocation of `closepl`. This is true of PCL and HP-GL Plotters, for example. Similarly, Plotters that normally output only a single page (PNM, GIF, Illustrator, and Fig Plotters) do so immediately after that page is ended. However, Postscript Plotters do not output their page(s) of graphics until they are deleted.

9.4.2 Object-drawing functions

The following are the "drawing functions" in `libplot`. When invoked on a Plotter, these functions cause it to draw objects (paths, circles, ellipses, points, markers, and text strings) on the associated graphics display. A *path* is a sequence of line segments, arc segments (either circular or elliptic), and/or Bezier curve segments (either quadratic or cubic). Paths are drawn incrementally, one segment at a time.

In the C binding, the names of all functions should be preceded by "pl_", unless the header file `plotcompat.h` is included. See Section 9.2.2 [C Compiling and Linking], page 88. In the C++ binding, these are member functions of the `Plotter` class and its subclasses.

int **`alabel`** (int *horiz_justify*, int *vert_justify*, const char **s*);

: `alabel` takes three arguments *horiz_justify*, *vert_justify*, and *s*, which specify an 'adjusted label,' i.e., a justified text string. The path under construction (if any) is ended, and the string *s* is drawn according to the specified justifications. If *horiz_justify* is equal to '`l`', '`c`', or '`r`', then the string will be drawn with left, center or right justification, relative to the current graphics cursor position. If *vert_justify* is equal to '`b`', '`x`', '`c`', or '`t`', then the bottom, baseline, center or top of the string will be placed even with the current graphics cursor position. The graphics cursor is moved to the right end of the string if left justification is specified, and to the left end if right justification is specified.

: The string may contain escape sequences of various sorts (see Section A.4 [Text String Format], page 131), though it should not contain line feeds or carriage returns. In fact it should include only printable characters, from the byte ranges `0x20`...`0x7e` and `0xa0`...`0xff`. The string may be plotted at a nonzero angle, if `textangle` has been called.

int **`arc`** (int *xc*, int *yc*, int *x0*, int *y0*, int *x1*, int *y1*);
int **`farc`** (double *xc*, double *yc*, double *x0*, double *y0*, double *x1*, double *y1*);
int **`arcrel`** (int *xc*, int *yc*, int *x0*, int *y0*, int *x1*, int *y1*);
int **`farcrel`** (double *xc*, double *yc*, double *x0*, double *y0*, double *x1*, double *y1*);

: `arc` and `farc` take six arguments specifying the beginning (*x0*, *y0*), end (*x1*, *y1*), and center (*xc*, *yc*) of a circular arc. If the graphics cursor is at (*x0*, *y0*) and a path is under construction, then the arc is added to the path. Otherwise the current path (if any) is ended, and the arc begins a new path. In all cases the graphics cursor is moved to (*x1*, *y1*).

: The direction of the arc (clockwise or counterclockwise) is determined by the convention that the arc, centered at (*xc*, *yc*), sweep through an angle of at most 180 degrees. If the three points appear to be collinear, the direction is taken to be counterclockwise. If (*xc*, *yc*) is not equidistant from (*x0*, *y0*) and (*x1*, *y1*) as it should be, it is corrected by being moved to the closest point on the perpendicular bisector of the line segment joining (*x0*, *y0*) and (*x1*, *y1*). `arcrel` and `farcrel` are similar to `arc` and `farc`, but use cursor-relative coordinates.

int **`bezier2`** (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*);
int **`fbezier2`** (double *x0*, double *y0*, double *x1*, double *y1*, double *x2*, double *y2*);
int **`bezier2rel`** (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*);
int **`fbezier2rel`** (double *x0*, double *y0*, double *x1*, double *y1*, double *x2*, double *y2*);

: `bezier2` and `fbezier2` take six arguments specifying the beginning `p0`=(*x0*, *y0*) and end `p2`=(*x2*, *y2*) of a quadratic Bezier curve, and its intermediate control point `p1`=(*x1*, *y1*). If the graphics cursor is at `p0` and a path is under construction, then the curve is added to the path. Otherwise the current path (if any) is ended, and the curve begins a new path. In all cases the graphics cursor is moved to `p2`. `bezier2rel` and `fbezier2rel` are similar to `bezier2` and `fbezier2`, but use cursor-relative coordinates.

The quadratic Bezier curve is tangent at p0 to the line segment joining p0 to p1, and is tangent at p2 to the line segment joining p1 to p2. So it fits snugly into a triangle with vertices p0, p1, and p2.

When using a PCL Plotter to draw Bezier curves on a LaserJet III, you should set the parameter PCL_BEZIERS to "no". That is because the LaserJet III, which was Hewlett–Packard's first PCL 5 printer, does not recognize the Bezier instructions supported by later PCL 5 printers. See Section 9.5 [Driver Parameters], page 118.

int `bezier3` (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*, int *x3*, int *y3*);
int `fbezier3` (double *x0*, double *y0*, double *x1*, double *y1*, double *x2*, double *y2*, double *x3*, double *y3*);
int `bezier3rel` (int *x0*, int *y0*, int *x1*, int *y1*, int *x2*, int *y2*, int *x3*, int *y3*);
int `fbezier3rel` (double *x0*, double *y0*, double *x1*, double *y1*, double *x2*, double *y2*, double *x3*, double *y3*);

bezier3 and fbezier3 take eight arguments specifying the beginning p0=(*x0*, *y0*) and end p3=(*x3*, *y3*) of a cubic Bezier curve, and its intermediate control points p1=(*x1*, *y1*) and p2=(*x2*, *y2*). If the graphics cursor is at p0 and a path is under construction, then the curve is added to the path. Otherwise the current path (if any) is ended, and the curve begins a new path. In all cases the graphics cursor is moved to p3. bezier3rel and fbezier3rel are similar to bezier3 and fbezier3, but use cursor-relative coordinates.

The cubic Bezier curve is tangent at p0 to the line segment joining p0 to p1, and is tangent at p3 to the line segment joining p2 to p3. So it fits snugly into a quadrangle with vertices p0, p1, p2, and p3.

When using a PCL Plotter to draw Bezier curves on a LaserJet III, you should set the parameter PCL_BEZIERS to "no". That is because the LaserJet III, which was Hewlett–Packard's first PCL 5 printer, does not recognize the Bezier instructions supported by later PCL 5 printers. See Section 9.5 [Driver Parameters], page 118.

int `box` (int *x1*, int y*1*, int *x2*, int *y2*);
int `fbox` (double *x1*, double *y1*, double *x2*, double *y2*);
int `boxrel` (int *x1*, int y*1*, int *x2*, int *y2*);
int `fboxrel` (double *x1*, double y*1*, double *x2*, double *y2*);

box and fbox take four arguments specifying the lower left corner (*x1*, *y1*) and upper right corner (*x2*, *y2*) of a 'box', or rectangle. The path under construction (if any) is ended, and the box is drawn as a new path. This path is also ended, and the graphics cursor is moved to the midpoint of the box. boxrel and fboxrel are similar to box and fbox, but use cursor-relative coordinates.

int `circle` (int *xc*, int *yc*, int *r*);
int `fcircle` (double *xc*, double *yc*, double *r*);
int `circlerel` (int *xc*, int *yc*, int *r*);
int `fcirclerel` (double *xc*, double *yc*, double *r*);

circle and fcircle take three arguments specifying the center (*xc*, *yc*) and radius (*r*) of a circle. The path under construction (if any) is ended, and the circle is drawn. The graphics cursor is moved to (*xc*, *yc*). `circlerel` and `fcirclerel` are similar to `circle` and `fcircle`, but use cursor-relative coordinates for *xc* and *yc*.

int `cont` (int *x*, int *y*);
int `fcont` (double *x*, double *y*);
int `contrel` (int *x*, int *y*);
int `fcontrel` (double *x*, double *y*);

`cont` and `fcont` take two arguments specifying the coordinates (*x*, *y*) of a point. If a path is under construction, the line segment from the current graphics cursor position to the point (*x*, *y*) is added to it. Otherwise the line segment begins a new path. In all cases the graphics cursor is moved to (*x*, *y*). `contrel` and `fcontrel` are similar to `cont` and `fcont`, but use cursor-relative coordinates.

int `ellarc` (int *xc*, int *yc*, int *x0*, int *y0*, int *x1*, int *y1*);
int `fellarc` (double *xc*, double *yc*, double *x0*, double *y0*, double *x1*, double *y1*);
int `ellarcrel` (int *xc*, int *yc*, int *x0*, int *y0*, int *x1*, int *y1*);
int `fellarcrel` (double *xc*, double *yc*, double *x0*, double *y0*, double *x1*, double *y1*);

`ellarc` and `fellarc` take six arguments specifying the three points `pc`=(*xc*,*yc*), `p0`=(*x0*,*y0*), and `p1`=(*x1*,*y1*) that define a so-called quarter ellipse. This is an elliptic arc from `p0` to `p1` with center `pc`. If the graphics cursor is at point `p0` and a path is under construction, the quarter-ellipse is added to it. Otherwise the path under construction (if any) is ended, and the quarter-ellipse begins a new path. In all cases the graphics cursor is moved to `p1`.

The quarter-ellipse is an affinely transformed version of a quarter circle. It is drawn so as to have control points `p0`, `p1`, and `p0`+`p1`−`pc`. This means that it is tangent at `p0` to the line segment joining `p0` to `p0`+`p1`−`pc`, and is tangent at `p1` to the line segment joining `p1` to `p0`+`p1`−`pc`. So it fits snugly into a triangle with these three control points as vertices. Notice that the third control point is the reflection of `pc` through the line joining `p0` and `p1`. `ellarcrel` and `fellarcrel` are similar to `ellarc` and `fellarc`, but use cursor-relative coordinates.

int `ellipse` (int *xc*, int *yc*, int *rx*, int *ry*, int *angle*);
int `fellipse` (double *xc*, double *yc*, double *rx*, double *ry*, double *angle*);
int `ellipserel` (int *xc*, int *yc*, int *rx*, int *ry*, int *angle*);
int `fellipserel` (double *xc*, double *yc*, double *rx*, double *ry*, double *angle*);

`ellipse` and `fellipse` take five arguments specifying the center (*xc*, *yc*) of an ellipse, the lengths of its semiaxes (*rx* and *ry*), and the inclination of the first semiaxis in the counterclockwise direction from the x axis in the user coordinate system. The path under construction (if any) is ended, and the ellipse is drawn. The graphics cursor is moved to (*xc*, *yc*). `ellipserel` and `fellipserel` are similar to `ellipse` and `fellipse`, but use cursor-relative coordinates.

int `endpath` ();

`endpath` terminates the path under construction, if any. A path is constructed by one or more successive calls to `cont`, `line`, `arc`, `ellarc`, `bezier2`, `bezier3`, and/or their floating-point counterparts. The path will also be terminated if any non-path object is drawn, if any path-related drawing attribute is set, or if `move` or `fmove` is invoked to set the cursor position. So `endpath` is almost redundant. However, if a Plotter plots objects in real time, calling `endpath` will ensure that a constructed path is drawn on the graphics display without delay.

int **`label`** (const char **s*);
: `label` takes a single string argument *s* and draws the text contained in *s* at the current graphics cursor position. The text is left justified, and the graphics cursor is moved to the right end of the string. This function is provided for backward compatibility; the function call `label`(*s*) is equivalent to `alabel`('l','x',*s*).

int **`labelwidth`** (const char **s*);
double **`flabelwidth`** (const char **s*);
: `labelwidth` and `flabelwidth` compute and return the width of a string in the current font, in the user coordinate system. The string is not plotted.

int **`line`** (int *x1*, int *y1*, int *x2*, int *y2*);
int **`fline`** (double *x1*, double y*1*, double *x2*, double *y2*);
int **`linerel`** (int *x1*, int y*1*, int *x2*, int *y2*);
int **`flinerel`** (double *x1*, double y*1*, double *x2*, double *y2*);
: `line` and `fline` take four arguments specifying the start point (*x1*, *y1*) and end point (*x2*, *y2*) of a line segment. If the graphics cursor is at (*x1*, *y1*) and a path is under construction, the line segment is added to it. Otherwise the path under construction (if any) is ended, and the line segment begins a new path. In all cases the graphics cursor is moved to (*x2*, *y2*). `linerel` and `flinerel` are similar to `line` and `fline`, but use cursor-relative coordinates.

int **`marker`** (int *x*, int *y*, int *type*, int *size*);
int **`fmarker`** (double *x*, double *y*, int *type*, double *size*);
int **`markerrel`** (int *x*, int *y*, int *type*, int *size*);
int **`fmarkerrel`** (double *x*, double *y*, int *type*, double *size*);
: `marker` and `fmarker` take four arguments specifying the location (*x*,*y*) of a marker symbol, its type, and its size in user coordinates. The path under construction (if any) is ended, and the marker symbol is plotted. The graphics cursor is moved to (*x*,*y*). `markerrel` and `fmarkerrel` are similar to `marker` and `fmarker`, but use cursor-relative coordinates for the position (*x*,*y*).

  Marker symbol types 0 through 31 are taken from a standard set, and marker symbol types 32 and above are interpreted as the index of a character in the current text font. See Section A.5 [Marker Symbols], page 144.

int **`point`** (int *x*, int *y*);
int **`fpoint`** (double *x*, double *y*);
int **`pointrel`** (int *x*, int *y*);
int **`fpointrel`** (double *x*, double *y*);
: `point` and `fpoint` take two arguments specifying the coordinates (*x*, *y*) of a point. The path under construction (if any) is ended, and the point is plotted. (Plotters that produce bitmaps draw points as single pixels. Other Plotters draw them as small solid circles, usually the smallest that can be plotted.) The graphics cursor is moved to (*x*, *y*). `pointrel` and `fpointrel` are similar to `point` and `fpoint`, but use cursor-relative coordinates.

9.4.3 Attribute-setting functions

The following are the "attribute functions" in `libplot`. When invoked on a Plotter, these functions set its drawing attributes, or save them or restore them. Path-related attributes include graphics cursor position, pen color, fill color, fill rule, line thickness, line style, cap style, join style, and miter limit. Text-related attributes include pen color, font name, font size, and text angle.

Setting any path-related drawing attribute automatically terminates the path under construction (if any), as if the `endpath` operation had been invoked.

In the C binding, the names of all functions should be preceded by "pl_", unless the header file `plotcompat.h` is included. See Section 9.2.2 [C Compiling and Linking], page 88. In the C++ binding, these are member functions of the `Plotter` class and its subclasses.

int `capmod` (const char **s*);

: `capmod` sets the cap mode (i.e., cap style) for all paths subsequently drawn on the graphics display. Recognized styles are "butt" (the default), "round", and "projecting". The three styles are visibly distinct only if the line thickness is fairly large. Butt caps do not extend beyond the end of the path. The other two kinds do, however. Round caps are filled semicircles, and projecting caps are filled rectangular regions that extend a distance equal to half the line width beyond the end of the path.

: PCL and HP-GL Plotters support a fourth cap mode, "triangular". Plotters other than PCL and HP-GL Plotters treat "triangular" as equivalent to "round".

: This function has no effect on Tektronix Plotters. Also, it has no effect on HP-GL Plotters if the parameter `HPGL_VERSION` is set to a value less than "2" (the default). See Section 9.5 [Driver Parameters], page 118.

int `color` (int *red*, int *green*, int *blue*);

: `color` is a convenience function. Calling `color` is equivalent to calling both `pencolor` and `fillcolor`, to set both the the pen color and fill color of all objects subsequently drawn on the graphics display. Note that the physical fill color depends also on the fill fraction, which is specified by calling `filltype`.

int `colorname` (const char **name*);

: `colorname` is a convenience function. Calling `colorname` is equivalent to calling both `pencolorname` and `fillcolorname`, to set both the the pen color and fill color of all objects subsequently drawn on the graphics display. Note that the physical fill color depends also on the fill fraction, which is specified by calling `filltype`.

int `fillcolor` (int *red*, int *green*, int *blue*);

: `fillcolor` sets the fill color of all objects subsequently drawn on the graphics display, using a 48-bit RGB color model. The arguments *red*, *green* and *blue* specify the red, green and blue intensities of the fill color. Each is an integer in the range `0x0000...0xffff`, i.e., 0...65535. The choice (0, 0, 0) signifies black, and the choice (65535, 65535, 65535) signifies white. Note that the physical fill color depends also on the fill fraction, which is specified by calling `filltype`.

int `fillcolorname` (const char **name*);

`fillcolorname` sets the fill color of all objects subsequently drawn on the graphics display to be *name*. For information on what color names are recognized, see Appendix B [Color Names], page 146. Unrecognized colors are interpreted as "black". Note that the physical fill color depends also on the fill fraction, which is specified by calling `filltype`.

int `fillmod` (const char **s*);

`fillmod` sets the fill mode, i.e., fill rule, for all objects subsequently drawn on the graphics display. The fill rule affects only filled, self-intersecting paths: it determines which points are 'inside'. Two rules are supported: "even-odd" (the default for all Plotters), and "nonzero-winding". For the distinction, see the *Postscript Language Reference Manual*. "alternate" is an alias for "even-odd" and "winding" is an alias for "nonzero-winding".

Fig Plotters do not support the "nonzero-winding" fill rule, because `xfig` itself does not support it. Also, HPGL Plotters do not support it if `HPGL_VERSION` is set to a value less than "2" (the default). See Section 9.5 [Driver Parameters], page 118.

The LaserJet III, which was Hewlett–Packard's first PCL 5 printer, did not support the nonzero-winding fill rule. However, all later PCL 5 printers from Hewlett–Packard support it.

int `filltype` (int *level*);

`filltype` sets the fill fraction for all subsequently drawn objects. A value of 0 for *level* indicates that objects should be unfilled, or transparent. This is the default. A value in the range `0x0001`...`0xffff`, i.e., 1...65535, indicates that objects should be filled. A value of 1 signifies 100% filling (the fill color will simply be the color specified by calling `fillcolor` or `fillcolorname`). If *level*=`0xffff`, the fill color will be white. Values between `0x0001` and `0xffff` are interpreted as specifying a desaturation, or gray level. For example, `0x8000` specifies 50% filling (the fill color will be intermediate between the color specified by calling `fillcolor` or `fillcolorname`, and white).

Tektronix Plotters do not support filling, and HP-GL Plotters support filling of arbitrary paths only if the parameter `HPGL_VERSION` is equal to "1.5" or "2" (the default). (If the version is "1" then only circles and rectangles aligned with the coordinate axes may be filled.) *Opaque* filling, including white filling, is supported only if the parameter `HPGL_VERSION` is "2" and the parameter `HPGL_OPAQUE_MODE` is "yes" (the default). See Section 9.5 [Driver Parameters], page 118.

int `fmiterlimit` (double *limit*);

`fmiterlimit` sets the miter limit for all paths subsequently drawn on the graphics display. The miter limit controls the treatment of corners, if the join mode is set to "miter" (the default). At a join point of a path, the 'miter length' is defined to be the distance between the inner corner and the outer corner. The miter limit is the maximum value that will be tolerated for the miter length divided by the line thickness. If this value is exceeded, the miter will be cut off: the "bevel" join mode will be used instead.

Examples of typical values for *limit* are 10.43 (the default, which cuts off miters if the join angle is less than 11 degrees), 2.0 (the same, for 60 degrees), and 1.414 (the same, for 90 degrees). In general, the miter limit is the cosecant of one-half the minimum angle for mitered joins. The minimum meaningful value for *limit* is 1.0, which converts all mitered joins to beveled joins, irrespective of join angle. Specifying a value less than 1.0 resets the limit to the default.

This function has no effect on X Drawable Plotters or X Plotters, since the X Window System miter limit, which is also 10.43, cannot be altered. It also has no effect on Tektronix Plotters or Fig Plotters, or on HP-GL Plotters if the parameter `HPGL_VERSION` is set to a value less than "2" (the default). See Section 9.5 [Driver Parameters], page 118.

int `fontname` (const char **font_name*);
double `ffontname` (const char **font_name*);

`fontname` and `ffontname` take a single case-insensitive string argument, *font_name*, specifying the name of the font to be used for all text strings subsequently drawn on the graphics display. (The font for plotting strings is fully specified by calling `fontname`, `fontsize`, and `textangle`.) The size of the font in user coordinates is returned.

The default font name depends on the type of Plotter. It is "Helvetica" for all Plotters except PNM, GIF, HP-GL, Tektronix and Metafile Plotters, for which it is "HersheySerif". If the argument *font_name* is NULL or the empty string, or the font is not available, the default font name will be used. Which fonts are available also depends on the type of Plotter; for a list of all available fonts, see Section A.1 [Text Fonts], page 124.

int `fontsize` (int *size*);
double `ffontsize` (double *size*);

`fontsize` and `ffontsize` take a single argument, interpreted as the size, in the user coordinate system, of the font to be used for all text strings subsequently drawn on the graphics display. (The font for plotting strings is fully specified by calling `fontname`, `fontsize`, and `textangle`.) The size of the font in user coordinates is returned. A negative value for *size* sets the size to the default, which depends on the type of Plotter. Typically, the default font size is 1/50 times the size (i.e., minimum dimension) of the display.

int `joinmod` (const char **s*);

`joinmod` sets the join mode (i.e., join style) for all paths subsequently drawn on the graphics display. Recognized styles are "miter" (the default), "round", and "bevel". The three styles are visibly distinct only if the line thickness is fairly large. Mitered joins are sharp, rounded joins are round, and beveled joins are squared off. However, unusually sharp joins are never mitered: instead, they are beveled. The angle at which beveling replaces mitering may be specified by calling `fmiterlimit`.

PCL and HP-GL Plotters support a fourth join mode, "triangular". Plotters other than PCL and HP-GL Plotters treat "triangular" as equivalent to "round".

This function has no effect on Tektronix Plotters. Also, it has no effect on HP-GL Plotters if the parameter `HPGL_VERSION` is set to a value less than `"2"` (the default). See Section 9.5 [Driver Parameters], page 118.

int **`linedash`** (int *n*, const int **dashes*, int *offset*);
int **`flinedash`** (int *n*, const double **dashes*, double *offset*);

`linedash` and `flinedash` set the line style for all paths, circles, and ellipses subsequently drawn on the graphics display. They provide much finer control of dash patterns than the `linemod` function (see below) provides. *dashes* should be an array of length *n*. Its elements, which should be positive, are interpreted as distances in the user coordinate system. Along any path, circle, or ellipse, the elements *dashes*[0]...*dashes*[*n*-1] alternately specify the length of a dash and the length of a gap between dashes. When the end of the array is reached, the reading of the array wraps around to the beginning. If the array is empty, i.e., *n* equals zero, there is no dashing: the drawn line is solid.

The *offset* argument specifies the 'phase' of the dash pattern relative to the start of the path. It is interpreted as the distance into the dash pattern at which the dashing should begin. For example, if *offset* equals zero then the path will begin with a dash, of length *dashes*[0] in user space. If *offset* equals *dashes*[0] then the path will begin with a gap of length *dashes*[1], and so forth. *offset* is allowed to be negative.

Not all Plotters fully support `linedash` and `flinedash`. HP-GL and PCL Plotters cannot dash with a nonzero offset, and in the dash patterns used by X and X Drawable Plotters, each dash and each gap has a maximum length of 255 pixels. `linedash` and `flinedash` have no effect on Tektronix and Fig Plotters, and they have no effect on HP-GL Plotters for which the parameter `HPGL_VERSION` is less than `"2"` (the default; see Section 9.5 [Driver Parameters], page 118).

Warning: If the map from the user coordinate system to the device coordinate system is not uniform, each dash in a dashed path should ideally be drawn on the graphics display with a length that depends on its direction. But currently, only Postscript Plotters do this. Other Plotters always draw any specified dash with the same length, irrespective of its direction. The length that is used is the minimum length, in the device coordinate system, that can correspond to the specified dash length in the user coordinate system.

int **`linemod`** (const char **s*);

`linemod` sets the line style for all paths, circles, and ellipses subsequently drawn on the graphics display. The supported line styles are `"solid"`, `"dotted"`, `"dotdashed"`, `"shortdashed"`, `"longdashed"`, `"dotdotdashed"`, `"dotdotdotdashed"`, and `"disconnected"`. The first seven correspond to the following dash patterns:

```
"solid"             --------------------------------
"dotted"            -   -   -   -   -   -   -   -
"dotdashed"         ----   -   ----   -   ----   -
"shortdashed"       ----    ----    ----    ----
"longdashed"        -------     -------     -------
"dotdotdashed"      ----   -   -   ----   -   -
"dotdotdotdashed"   ----   -   -   -   ----   -   -   -
```

In the preceding patterns, each hyphen stands for one line thickness. This is the case for sufficiently thick lines, at least. So for sufficiently thick lines, the distance over which a dash pattern repeats is scaled proportionately to the line thickness.

The "disconnected" line style is special. A "disconnected" path is rendered as a set of filled circles, each of which has diameter equal to the nominal line thickness. One of these circles is centered on each of the juncture points of the path (i.e., the endpoints of the line segments or arcs from which it is constructed). Circles and ellipses with "disconnected" line style are invisible. Disconnected paths, circles, and ellipses are not filled.

All line styles are supported by all Plotters, with the following exceptions. HP-GL Plotters do not support the "dotdotdotdashed" style unless the parameter `HPGL_VERSION` is set to "2" (the default). Tektronix Plotters do not support the "dotdotdotdashed" style, and do not support the "dotdotdashed" style unless the parameter `TERM` is set to "kermit". See Section 9.5 [Driver Parameters], page 118.

int **`linewidth`** (int *size*);
int **`flinewidth`** (double *size*);

`linewidth` and `flinewidth` set the thickness, in the user coordinate system, of all paths, circles, and ellipses subsequently drawn on the graphics display. A negative value resets the thickness to the default. The default thickness depends on the type of Plotter. For most Plotters, it is 1/850 times the size (i.e., minimum dimension) of the display. But for Plotters that produce bitmaps, i.e., X Plotters, X Drawable Plotters, PNM Plotters, and GIF Plotters, it is zero.

By convention, a zero-thickness line is the thinnest line that can be drawn. However, the drawing editors `idraw` and `xfig` treat zero-thickness lines as invisible. So when producing editable graphics with a Postscript or Fig Plotter, using a zero line thickness may not be desirable.

Tektronix Plotters do not support drawing with other than a default thickness, and HP-GL Plotters do not support doing so if the parameter `HPGL_VERSION` is set to a value less than "2" (the default; see Section 9.5 [Driver Parameters], page 118).

Warning: If the map from the user coordinate system to the device coordinate system is not uniform, each line segment in a polygonal path should ideally be drawn on the graphics display with a thickness that depends on its direction. But currently, only Postscript Plotters do this. Other Plotters draw all line segments in a path with the same thickness. The thickness that is used is the minimum thickness, in the device coordinate system, that can correspond to the thickness of the path in the user coordinate system.

int `move` (int *x*, int *y*);
int `fmove` (double *x*, double *y*);
int `moverel` (int *x*, int *y*);
int `fmoverel` (double *x*, double *y*);

`move` and `fmove` take two arguments specifying the coordinates (*x*, *y*) of a point to which the graphics cursor should be moved. The path under construction (if any) is ended, and the graphics cursor is moved to (*x*, *y*). This is equivalent to lifting the pen on a plotter and moving it to a new position, without drawing any line. `moverel` and `fmoverel` are similar to `move` and `fmove`, but use cursor-relative coordinates.

int `pencolor` (int *red*, int *green*, int *blue*);

`pencolor` sets the pen color of all objects subsequently drawn on the graphics display, using a 48-bit RGB color model. The arguments *red*, *green* and *blue* specify the red, green and blue intensities of the pen color. Each is an integer in the range `0x0000`...`0xffff`, i.e., 0...65535. The choice (0, 0, 0) signifies black, and the choice (65535, 65535, 65535) signifies white.

HP-GL Plotters support drawing with a white pen only if the value of the parameter `HPGL_VERSION` is "2" (the default), and the value of the parameter `HPGL_OPAQUE_MODE` is "yes" (the default). See Section 9.5 [Driver Parameters], page 118.

int `pencolorname` (const char **name*);

`pencolorname` sets the pen color of all objects subsequently drawn on the graphics display to be *name*. For information on what color names are recognized, see Appendix B [Color Names], page 146. Unrecognized colors are interpreted as "black".

HP-GL Plotters support drawing with a white pen only if the value of the parameter `HPGL_VERSION` is "2" (the default) and the value of the parameter `HPGL_OPAQUE_MODE` is "yes" (the default). See Section 9.5 [Driver Parameters], page 118.

int `restorestate` ();

`restorestate` pops the current graphics context off the stack of drawing states. The graphics context consists largely of `libplot`'s drawing attributes, which are set by the attribute functions documented in this section. So popping off the graphics context restores the drawing attributes to values they previously had. A path under construction is regarded as part of the graphics context. For this reason, calling `restorestate` automatically calls `endpath` to terminate the path under construction, if any. All graphics contexts on the stack are popped off when `closepl` is called, as if `restorestate` had been called repeatedly.

int `savestate` ();

`savestate` pushes the current graphics context onto the stack of drawing states. The graphics context consists largely of `libplot`'s drawing attributes, which are set by the attribute functions documented in this section. A path under construction, if any, is regarded as part of the graphics context. That is because paths may be drawn incrementally, one line segment or arc at a time. When a graphics context is returned to, the path under construction may be continued.

int `textangle` (int *angle*);
double `ftextangle` (double *angle*);

`textangle` and `ftextangle` take one argument, which specifies the angle in degrees counterclockwise from the x (horizontal) axis in the user coordinate system, for text strings subsequently drawn on the graphics display. The default angle is zero. (The font for plotting strings is fully specified by calling `fontname`, `fontsize`, and `textangle`.) The size of the font for plotting strings, in user coordinates, is returned.

9.4.4 Mapping functions

The following are the "mapping functions" in `libplot`. When invoked on a Plotter, these functions affect the affine transformation it employs for mapping from the user coordinate system to the device coordinate system. They may be viewed as performing transformations of the user coordinate system. Their names resemble those of the corresponding functions in the Postscript language. For information on how to use them to draw graphics efficiently, consult any good book on Postscript programming, or the *Postscript Language Reference Manual*.

In the C binding, the names of all functions should be preceded by "pl_", unless the header file `plotcompat.h` is included. See Section 9.2.2 [C Compiling and Linking], page 88. In the C++ binding, these are member functions of the `Plotter` class and its subclasses.

int `fconcat` (double *m0*, double *m1*, double *m2*, double *m3*, double *tx*, double *ty*);

Apply a Postscript-style transformation matrix, i.e., affine map, to the user coordinate system. That is, apply the linear transformation defined by the two-by-two matrix [*m0 m1 m2 m3*] to the user coordinate system, and also translate by *tx* units in the x direction and *ty* units in the y direction, relative to the former user coordinate system. The following three functions (`frotate`, `fscale`, `ftranslate`) are convenience functions that are special cases of `fconcat`.

int `frotate` (double *theta*);

Rotate the user coordinate system axes about their origin by *theta* degrees, with respect to their former orientation. The position of the user coordinate origin and the size of the x and y units remain unchanged.

int `fscale` (double *sx*, double *sy*);

Make the x and y units in the user coordinate system be the size of *sx* and *sy* units in the former user coordinate system. The position of the user coordinate origin and the orientation of the coordinate axes are unchanged.

int `ftranslate` (double *tx*, double *ty*);

Move the origin of the user coordinate system by *tx* units in the x direction and *ty* units in the y direction, relative to the former user coordinate system. The size of the x and y units and the orientation of the coordinate axes are unchanged.

9.5 Device driver parameters

In designing the `libplot` library, every effort was made to make the Plotter interface independent of the type of Plotter. To the extent that device dependence exists, it is captured by a manageable number of *device driver parameters*.

In the C binding, a value for any parameter may be specified by calling the `pl_parampl` function. The `pl_parampl` function does not operate on any particular Plotter: it belongs to the C binding as a whole. The parameter values used by any Plotter are constant over the lifetime of the Plotter, and are those that were in effect when the Plotter was created. Each driver parameter has a value that is allowed to be a generic pointer (a `void *`). For most parameters, this value should be a string (a `char *`). `pl_parampl` may be called any number of times. A parameter may be unset by calling `pl_parampl` with a value argument of NULL.

If at Plotter creation time a parameter is *not* set, its default value will be used, unless the parameter is string-valued and there is an environment variable of the same name, in which case the value of that environment variable will be used. This rule increases run-time flexibility: an application programmer may allow non-critical driver parameters to be specified by the user via environment variables.

In the C++ binding, the function `Plotter::parampl` is the analogue of `pl_parampl`. It is a static member of the `Plotter` class and its subclasses. Like `pl_parampl`, it does not act on any particular Plotter: rather, it sets the parameters of subsequently created Plotters.

The following are the currently recognized parameters (unrecognized ones are ignored). The most important ones are `DISPLAY`, which affects X Plotters, `BITMAPSIZE`, which affects X Plotters, PNM Plotters, and GIF Plotters, and `PAGESIZE`, which affects Illustrator, Postscript, Fig, and HP-GL Plotters. These three parameters are listed first and the others alphabetically. Many of the parameters, such as the several whose names begin with "HPGL", affect only a single type of Plotter.

`DISPLAY`
: (Default NULL.) The X Window System display on which the graphics display will be popped up, as an X window. This is relevant only to X Plotters.

`BITMAPSIZE`
: (Default "570x570".) The size of the graphics display in terms of pixels. This is relevant only to X Plotters, PNM Plotters, and GIF Plotters. For X Plotters, the value of this parameter will automatically, if it is not set, be taken from the X resource `Xplot.geometry`. This is for backward compatibility.

`PAGESIZE`
: (Default "letter".) The size of the page on which the graphics display will be positioned. This is relevant only to Illustrator, Postscript, Fig, PCL, and HP-GL Plotters. "letter" means an 8.5 in by 11 in page. Any ISO page size in the range "a0"..."a4" or ANSI page size in the range "a"..."e" may be specified ("letter" is an alias for "a" and "tabloid" is an alias for "b"). "legal", "ledger", and "b5" are recognized page sizes also.

  For Illustrator and Postscript Plotters, the graphics display will be a square region centered on the specified page and occupying its full width, with allowance being made for margins. For Fig Plotters, the graphics display will be a square region of the same size, positioned in the upper left corner of an xfig display. For PCL and HP-GL Plotters, the graphics display will be a square region of the same size, but may be positioned differently. For PCL Plotters, fine control over its positioning on the page may be accomplished by setting the PCL_XOFFSET and PCL_YOFFSET parameters. For HP-GL Plotters, HPGL_XOFFSET and HPGL_YOFFSET are used similarly.

BG_COLOR
: (Default "white".) The initial background color of the graphics display, when drawing each page of graphics. This is relevant to X Plotters, PNM Plotters, GIF Plotters, and X Drawable Plotters (for the last, the background color shows up only if erase is invoked). The background color may be changed at any later time by invoking the bgcolor (or bgcolorname) and erase operations. An unrecognized color name sets the background color to the default. For information on what names are recognized, see Appendix B [Color Names], page 146.

GIF_ANIMATION
: (Default "yes".) Relevant only to GIF Plotters. "yes" means that the erase operation will have special semantics: with the exception of its first invocation, it will act as a separator between successive images in the written-out pseudo-GIF file. "no" means that erase should act as it does on other Plotters that do not write graphics in real time, i.e., it should erase the image under construction by filling it with the background color. If "no" is specified, the pseudo-GIF file will contain only a single image.

GIF_DELAY
: (Default "0".) Relevant only to GIF Plotters. The delay, in hundredths of a second, after each image in a written-out animated pseudo-GIF file. The value should be an integer in the range "0"..."65535".

GIF_ITERATIONS
: (Default "0".) Relevant only to GIF Plotters. The number of times that an animated pseudo-GIF file should be 'looped'. The value should be an integer in the range "0"..."65535".

HPGL_ASSIGN_COLORS
: (Default "no".) Relevant only to HP-GL Plotters, and only if the value of HPGL_VERSION is "2". "no" means to draw with a fixed set of pens, specified by setting the HPGL_PENS parameter. "yes" means that pen colors will not restricted to the palette specified in HPGL_PENS: colors will be assigned to "logical pens" in the range #1...#31, as needed. Other than color LaserJet printers and DesignJet plotters, not many HP-GL/2 devices allow the assignment of colors to logical pens. So this parameter should be used with caution.

HPGL_OPAQUE_MODE
: (Default "yes".) Relevant only to HP-GL Plotters, and only if the value of HPGL_VERSION is "2". "yes" means that the HP-GL/2 output device should be switched into opaque mode, rather than transparent mode. This allows objects to be filled with opaque white and other opaque colors. It also allows the drawing of visible white lines, which by convention are drawn with pen #0. Not all HP-GL/2 devices support opaque mode or the use of pen #0 to draw

visible white lines. In particular, HP-GL/2 pen plotters do not. Some older HP-GL/2 devices reportedly malfunction if asked to switch into opaque mode. If the output of an HP-GL Plotter is to be sent to such a device, a "no" value is recommended.

`HPGL_PENS`
: (Default "1=black:2=red:3=green:4=yellow:5=blue:6=magenta:7=cyan" if the value of `HPGL_VERSION` is "1.5" or "2" and "1=black" if the value of `HPGL_VERSION` is "1". Relevant only to HP-GL Plotters. The set of available pens; the format should be self-explanatory. The color for any pen in the range #1. . .#31 may be specified. For information on what color names are recognized, see Appendix B [Color Names], page 146. Pen #1 must always be present, though it need not be black. Any other pen in the range #1. . .#31 may be omitted.

`HPGL_ROTATE`
: (Default "0".) Relevant only to HP-GL Plotters. The angle, in degrees, by which the graphics display should be rotated on the page relative to the default orientation. Recognized values are "0", "90", "180", and "270"; "no" and "yes" are equivalent to "0" and "90" respectively. This parameter is provided to facilitate switching between portrait and landscape orientations. For HP-GL devices this is frequently a concern, since some HP-GL devices ("plotters") draw with a default landscape orientation, and others ("printers") draw with a default portrait orientation. "180" and "270" are supported only if `HPGL_VERSION` is "2".

`HPGL_VERSION`
: (Default "2".) Relevant only to HP-GL Plotters. "1" means that the output should be generic HP-GL, "1.5" means that the output should be suitable for the HP7550A graphics plotter and the HP758x, HP7595A and HP7596A drafting plotters (HP-GL with some HP-GL/2 extensions), and "2" means that the output should be modern HP-GL/2. If the version is less than "2" then the only available fonts will be vector fonts, and all paths will be drawn with a default thickness, so that invoking `capmod`, `joinmod`, and `fmiterlimit` will have no effect. Also, the 'nonzero winding number rule' will not be supported when filling paths, so invoking `fillmod` will have no effect. Additionally, if the version is "1" then the filling of arbitrary paths will not be supported (circles and rectangles aligned with the coordinate axes may be filled, however).

`HPGL_XOFFSET, HPGL_YOFFSET`
: (Defaults "0.0cm" and "0.0cm".) Relevant only to HP-GL Plotters. Adjustments, in the x and y directions, of the position of the graphics display on the page. They may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in".

`INTERLACE`
: (Default "no".) Relevant only to GIF Plotters. If the value is "yes", the pseudo-GIF output file will be interlaced. That means that it will be displayed in an interlaced (nonlinear) way by many applications.

MAX_LINE_LENGTH
: (Default "500".) The maximum number of points that a path may contain, before it is flushed to the display device. If this flushing occurs, the path will be split into two or more sub-paths, though the splitting should not be noticeable. Splitting will not be performed if the path is filled.

  This parameter is relevant to all Plotters except Tektronix and Metafile Plotters. The reason for splitting long paths is that some display devices (e.g., old Postscript printers and HP-GL pen plotters) have limited buffer sizes. It is not relevant to Tektronix or Metafile Plotters, since they draw paths in real time and have no buffer limitations.

META_PORTABLE
: (Default "no".) Relevant only to Metafile Plotters. "yes" means that the output should be in a portable (human-readable) version of the metafile format, rather than the default (binary) version. See Appendix C [Metafiles], page 147.

PCL_ASSIGN_COLORS
: (Default "no".) Relevant only to PCL Plotters. "no" means to draw with a fixed set of pens. "yes" means that pen colors will not restricted to this palette: colors will be assigned to "logical pens", as needed. Other than color LaserJet printers, not many PCL 5 devices allow the assignment of colors to logical pens. So this parameter should be used with caution.

PCL_BEZIERS
: (Default "yes".) Relevant only to PCL Plotters. "yes" means that when drawing Bezier curves, the special 'Bezier instructions' will be used. "no" means that these instructions will not be used. Instead, each Bezier curve will be approximated and drawn as a polygonal line. Other than the LaserJet III, which was Hewlett–Packard's first PCL 5 printer, all Hewlett–Packard's PCL 5 printers support the Bezier instructions.

PCL_ROTATE
: (Default "0".) Relevant only to PCL Plotters. The angle, in degrees, by which the graphics display should be rotated on the page relative to the default orientation. Recognized values are "0", "90", "180", and "270"; "no" and "yes" are equivalent to "0" and "90" respectively. This parameter is provided to facilitate switching between portrait and landscape orientations. For PCL 5 devices this is frequently a concern, since some PCL 5 devices ("plotters") draw with a default landscape orientation, and others ("printers") draw with a default portrait orientation.

PCL_XOFFSET, PCL_YOFFSET
: (Defaults "0.0cm" and "0.0cm".) Relevant only to PCL Plotters. Adjustments, in the x and y directions, of the position of the graphics display on the page. They may be specified in centimeters, millimeters, or inches. For example, an offset could be specified as "2cm" or "1.2in".

PNM_PORTABLE
: (Default "no".) Relevant only to PNM Plotters. "yes" means that the output should be in a portable (human-readable) version of PBM/PGM/PPM format,

rather than the default (binary) version. 'Portable' is something of a misnomer, since binary PBM/PGM/PPM files are also portable, in the sense that they are machine-independent.

`TERM` (Default NULL.) Relevant only to Tektronix Plotters. If the value is "xterm", "xterms", or "kterm", it is taken as a sign that the current application is running in an X Window System VT100 terminal emulator: an `xterm`. Before drawing graphics, a Tektronix Plotter will emit an escape sequence that causes the terminal emulator's auxiliary Tektronix window, which is normally hidden, to pop up. After the graphics are drawn, an escape sequence that returns control to the original VT100 window will be emitted. The Tektronix window will remain on the screen.

If the value is "kermit", "ansi.sys", "ansissys", "ansi.sysk", or "ansisysk", it is taken as a sign that the current application is running in the VT100 terminal emulator provided by the MS-DOS version of `kermit`. Before drawing graphics, a Tektronix Plotter will emit an escape sequence that switches the terminal emulator to Tektronix mode. Also, some of the Tektronix control codes emitted by the Plotter will be `kermit`-specific. There will be a limited amount of color support, which is not normally the case (the 16 `ansi.sys` colors will be supported). The "dotdotdashed" line style will be supported, which is also not normally the case. After drawing graphics, the Plotter will emit an escape sequence that returns the emulator to VT100 mode. The key sequence 'ALT minus' may be employed manually within `kermit` to switch between the two modes.

`TRANSPARENT_COLOR`

(Default "none".) Relevant only to GIF Plotters. If the value is a recognized color name, that color, if it appears in the pseudo-GIF output file, will be treated as transparent by most applications that display GIF files. For information on what names are recognized, see Appendix B [Color Names], page 146.

If `TRANSPARENT_COLOR` is set and an animated pseudo-GIF file is produced, the 'restore to background' disposal method will be used for each image in the file. Otherwise, the 'unspecified' disposal method will be used.

`USE_DOUBLE_BUFFERING`

(Default "no".) Relevant only to X Plotters and X Drawable Plotters. If the value is "yes", a double buffering scheme will be used when drawing graphics. Each frame of graphics, within a `openpl`...`closepl` pair, will be written to an off-screen buffer rather than to the Plotter's display. When `erase` is invoked to end a frame, or when `closepl` is invoked, the contents of the off-screen buffer will be copied to the Plotter's display, pixel by pixel. If successive frames differ only slightly, this will create the illusion of smooth animation.

Some X displays provide special hardware support for double buffering. If this support is available, the X Plotter will detect its presence, and will draw graphics using the appropriate extension to the X11 protocol (either DBE or MBX). In this case the animation will be significantly faster; on high-end graphics hardware, at least.

VANISH_ON_DELETE
: (Default "no".) Relevant only to X Plotters. If the value is "yes", when a Plotter is deleted, the window or windows that it has popped up will vanish. Otherwise, each such window will remain on the screen until it is removed by the user (by typing 'q' in it, or by clicking with a mouse).

XDRAWABLE_COLORMAP
: (Default NULL.) Relevant only to X Drawable Plotters. If the value is non-NULL, it should be a `Colormap *`, a pointer to a colormap from which colors should be allocated. NULL indicates that the colormap to be used should be the default colormap of the default screen of the X display.

XDRAWABLE_DISPLAY
: (Default NULL.) Relevant only to X Drawable Plotters. The value should be a `Display *`, a pointer to the X display with which the drawable(s) to be drawn in are associated.

XDRAWABLE_DRAWABLE1
XDRAWABLE_DRAWABLE2
: (Default NULL.) Relevant only to X Drawable Plotters. If set, the value of each of these parameters should be a `Drawable *`, a pointer to a drawable to be drawn in. A 'drawable' is either a window or a pixmap. At the time an X Drawable Plotter is created, at least one of the two parameters must be set.

  X Drawable Plotters support simultaneous drawing in two drawables because it is often useful to be able to draw graphics simultaneously in both an X window and its background pixmap. If two drawables are specified, they must have the same dimensions and depth, and be associated with the same screen of the X display.

Appendix A Fonts, Strings, and Symbols

The `libplot` vector graphics library and applications built on it, such as `graph` and `plot`, can draw text strings in a wide variety of fonts. Text strings may include characters from more than one font in a typeface, and may include superscripts, subscripts, and square roots. A wide variety of plotting symbols can also be drawn. The following sections explain how to use these features.

A.1 Available text fonts

The `libplot` library and applications built on it, such as `graph`, `plot`, `tek2plot`, and `pic2plot`, can use many fonts. These include 22 Hershey vector fonts, 35 Postscript fonts, 45 PCL 5 fonts, and 18 Hewlett–Packard vector fonts. We call these 120 supported fonts the 'built-in' fonts. The Hershey fonts are constructed from stroked characters digitized c. 1967 by Dr. Allen V. Hershey at the U.S. Naval Surface Weapons Center in Dahlgren, VA. The 35 Postscript fonts are the outline fonts resident in all modern Postscript printers, and the 45 PCL 5 fonts are the outline fonts resident in modern Hewlett–Packard LaserJet printers and plotters. (The old LaserJet III, which was Hewlett–Packard's first PCL 5 printer, supported only 8 of the 45.) The 18 Hewlett–Packard vector fonts are fonts that are resident in Hewlett–Packard printers and plotters (mostly the latter).

The Hershey fonts can be used by all types of Plotter supported by `libplot`, and the Postscript fonts can be used by X, Illustrator, Postscript, and Fig Plotters. So, for example, all variants of `graph` can use the Hershey fonts, and `graph -T X`, `graph -T ai`, `graph -T ps` and `graph -T fig` can use the Postscript fonts. The PCL 5 fonts can be used by by Illustrator, PCL, and HP-GL Plotters, and by `graph -T ai`, `graph -T pcl`, and `graph -T hpgl`. The Hewlett–Packard vector fonts can be used by by PCL and HP-GL Plotters, and by `graph -T pcl` and `graph -T hpgl`. X Plotters and `graph -T X` are not restricted to the built-in Hershey and Postscript fonts. They can use any X Window System font.

The `plotfont` utility, which accepts the '`-T`' option, will print a character map of any font that is available in the specified output format. See Chapter 6 [plotfont], page 50.

For the purpose of plotting text strings (see Section A.4 [Text String Format], page 131), the 120 built-in fonts are divided into typefaces. As you can see from the following tables, our convention is that in any typeface with more than a single font, font #1 is the normal font, font #2 is italic or oblique, font #3 is bold, and font #4 is bold italic or bold oblique. Additional variants (if any) are numbered #5 and higher.

The 22 Hershey fonts are divided into typefaces as follows.

- HersheySerif
  1. HersheySerif
  2. HersheySerif-Italic
  3. HersheySerif-Bold
  4. HersheySerif-BoldItalic
  5. HersheyCyrillic
  6. HersheyCyrillic-Oblique

7. HersheyEUC

- HersheySans
  1. HersheySans
  2. HersheySans-Oblique
  3. HersheySans-Bold
  4. HersheySans-BoldOblique
- HersheyScript
  1. HersheyScript
  2. HersheyScript
  3. HersheyScript-Bold
  4. HersheyScript-Bold
- HersheyGothicEnglish
- HersheyGothicGerman
- HersheyGothicItalian
- HersheySerifSymbol
  1. HersheySerifSymbol
  2. HersheySerifSymbol-Oblique
  3. HersheySerifSymbol-Bold
  4. HersheySerifSymbol-BoldOblique
- HersheySansSymbol
  1. HersheySansSymbol
  2. HersheySansSymbol-Oblique

Nearly all Hershey fonts except the Symbol fonts use the ISO-Latin-1 encoding, which is a superset of ASCII. The Symbol fonts consist of Greek characters and mathematical symbols, and use the symbol font encoding documented in the *Postscript Language Reference Manual*. By convention, each Hershey typeface contains a symbol font (HersheySerifSymbol or HersheySansSymbol, as appropriate) as font #0.

HersheyCyrillic, HersheyCyrillic-Oblique, and HersheyEUC (which is a Japanese font) are the only non-Symbol Hershey fonts that do not use the ISO-Latin-1 encoding. For their encodings, see Section A.2 [Cyrillic and Japanese], page 129.

The 35 Postscript fonts are divided into typefaces as follows.

- Helvetica
  1. Helvetica
  2. Helvetica-Oblique
  3. Helvetica-Bold
  4. Helvetica-BoldOblique
- Helvetica-Narrow
  1. Helvetica-Narrow
  2. Helvetica-Narrow-Oblique

   3. Helvetica-Narrow-Bold
   4. Helvetica-Narrow-BoldOblique
- Times
   1. Times-Roman
   2. Times-Italic
   3. Times-Bold
   4. Times-BoldItalic
- AvantGarde
   1. AvantGarde-Book
   2. AvantGarde-BookOblique
   3. AvantGarde-Demi
   4. AvantGarde-DemiOblique
- Bookman
   1. Bookman-Light
   2. Bookman-LightItalic
   3. Bookman-Demi
   4. Bookman-DemiItalic
- Courier
   1. Courier
   2. Courier-Oblique
   3. Courier-Bold
   4. Courier-BoldOblique
- NewCenturySchlbk
   1. NewCenturySchlbk-Roman
   2. NewCenturySchlbk-Italic
   3. NewCenturySchlbk-Bold
   4. NewCenturySchlbk-BoldItalic
- Palatino
   1. Palatino-Roman
   2. Palatino-Italic
   3. Palatino-Bold
   4. Palatino-BoldItalic
- ZapfChancery-MediumItalic
- ZapfDingbats
- Symbol

All Postscript fonts except the ZapfDingbats and Symbol fonts use the ISO-Latin-1 encoding. The encodings used by the ZapfDingbats and Symbol fonts are documented in the *Postscript Language Reference Manual*. By convention, each Postscript typeface contains the Symbol font as font #0.

The 45 PCL 5 fonts are divided into typefaces as follows.

- Univers
  1. Univers
  2. Univers-Oblique
  3. Univers-Bold
  4. Univers-BoldOblique
- UniversCondensed
  1. UniversCondensed
  2. UniversCondensed-Oblique
  3. UniversCondensed-Bold
  4. UniversCondensed-BoldOblique
- CGTimes
  1. CGTimes-Roman
  2. CGTimes-Italic
  3. CGTimes-Bold
  4. CGTimes-BoldItalic
- Albertus
  1. AlbertusMedium
  2. AlbertusMedium
  3. AlbertusExtraBold
  4. AlbertusExtraBold
- AntiqueOlive
  1. AntiqueOlive
  2. AntiqueOlive-Italic
  3. AntiqueOlive-Bold
- Arial
  1. Arial-Roman
  2. Arial-Italic
  3. Arial-Bold
  4. Arial-BoldItalic
- ClarendonCondensed
- Coronet
- Courier
  1. Courier
  2. Courier-Italic
  3. Courier-Bold
  4. Courier-BoldItalic
- Garamond
  1. Garamond

   2. Garamond-Italic
   3. Garamond-Bold
   4. Garamond-BoldItalic
- LetterGothic
   1. LetterGothic-Roman
   2. LetterGothic-Italic
   3. LetterGothic-Bold
   4. LetterGothic-BoldItalic
- Marigold
- CGOmega
   1. CGOmega-Roman
   2. CGOmega-Italic
   3. CGOmega-Bold
   4. CGOmega-BoldItalic
- TimesNewRoman
   1. TimesNewRoman
   2. TimesNewRoman-Italic
   3. TimesNewRoman-Bold
   4. TimesNewRoman-BoldItalic
- Wingdings
- Symbol

All PCL 5 fonts except the Wingdings and Symbol fonts use the ISO-Latin-1 encoding. The encoding used by the Symbol font is the symbol font encoding documented in the *Postscript Language Reference Manual*. By convention, each PCL typeface contains the Symbol font as font #0.

The 18 Hewlett–Packard vector fonts are divided into typefaces as follows.

- Arc
   1. Arc
   2. Arc-Oblique
   3. Arc-Bold
   4. Arc-BoldOblique
- Stick
   1. Stick
   2. Stick-Oblique
   3. Stick-Bold
   4. Stick-BoldOblique
- ArcANK
   1. ArcANK*

   2. ArcANK-Oblique*
   3. ArcANK-Bold*
   4. ArcANK-BoldOblique*
- StickANK
   1. StickANK*
   2. StickANK-Oblique*
   3. StickANK-Bold*
   4. StickANK-BoldOblique*
- ArcSymbol*
- StickSymbol*

The Hewlett–Packard vector fonts with an asterisk (the ANK and Symbol fonts) are only available when producing HP-GL output for the HP7550A graphics plotter and the HP758x, HP7595A and HP7596A drafting plotters. The ANK fonts are Japanese fonts (see Section A.2 [Cyrillic and Japanese], page 129), and the Symbol fonts contain a few miscellaneous mathematical symbols. To ensure that these fonts are available, you must set the environment variable or driver parameter `HPGL_VERSION` to "1.5".

All Hewlett–Packard vector fonts except the ANK and Symbol fonts use the ISO-Latin-1 encoding. The Arc fonts are proportional (variable-width) fonts, and the Stick fonts are fixed-width fonts. If `HPGL_VERSION` is "1.5" then the Arc fonts will be kerned. But if `HPGL_VERSION` is "2" (the default), there will be no kerning. Apparently Hewlett–Packard dropped support for device-resident kerning tables when moving from HP-GL to modern HP-GL/2 and PCL 5. For information about Hewlett–Packard vector fonts and the way in which they are kerned (in pen plotters, at least), see the article by L. W. Hennessee et al. in the Nov. 1981 issue of the *Hewlett–Packard Journal*.

To what extent do the fonts supported by `libplot` contain ligatures? The Postscript fonts, the PCL 5 fonts, and the Hewlett–Packard vector fonts, at least as implemented in `libplot`, do not contain ligatures. However, six of the 22 Hershey fonts contain ligatures. The character combinations "fi", "ff", "fl", "ffi", and "ffl" are automatically drawn as ligatures in HersheySerif and HersheySerif-Italic. (Also in the two HersheyCyrillic fonts and HersheyEUC, since insofar as printable ASCII characters are concerned, they are identical [or almost identical] to HersheySerif.) In addition, "tz" and "ch" are ligatures in Hershey-GothicGerman. The German double-s character 'ß', which is called an 'eszet', is not treated as a ligature in any font. To obtain an eszet, you must either request one with the escape sequence "\ss" (see Section A.4 [Text String Format], page 131), or, if you have an 8-bit keyboard, type an eszet explicitly.

A.2 Cyrillic and Japanese fonts

The built-in fonts discussed in the previous section include Cyrillic and Japanese vector fonts. This section explains how these fonts are encoded, i.e., how their character maps are laid out. You may use the `plotfont` utility to display the character map for any font, including the Cyrillic and Japanese vector fonts. See Chapter 6 [plotfont], page 50.

The HersheyCyrillic and HersheyCyrillic-Oblique fonts use an encoding called KOI8-R, a superset of ASCII that has become the de facto standard for Unix and networking applications in the former Soviet Union. Insofar as printable ASCII characters go, they resemble the HersheySerif vector font. But their upper halves are different. The byte range `0xc0`...`0xdf` contains lower-case Cyrillic characters and the byte range `0xe0`...`0xff` contains upper case Cyrillic characters. Additional Cyrillic characters are located at `0xa3` and `0xb3`. For more on the encoding scheme, see http://www.nagual.pp.ru/~ache/koi8.html, the official KOI8-R Web page and Internet RFC 1489, which is available from the http://www.isi.edu, Information Sciences Institute.

The HersheyEUC font is a vector font that is is used for displaying Japanese text. It uses the 8-bit EUC-JP encoding. EUC stands for 'extended Unix code', which is a scheme for encoding Japanese, and also other character sets (e.g., Greek and Cyrillic) as multibyte character strings. The format of EUC strings is explained in Ken Lunde's *Understanding Japanese Information Processing* (O'Reilly, 1993), which contains much additional information on Japanese text processing. See also http://www.ora.com/people/authors/lunde/cjk_inf.html, his on-line supplement.

In the HersheyEUC font, characters in the printable ASCII range, `0x20`...`0x7e`, are similar to HersheySerif (their encoding is 'JIS Roman', an ASCII variant standardized by the Japanese Industrial Standards Committee). Also, each successive pair of bytes in the `0xa1`...`0xfe` range defines a single character in the JIS X0208 standard. The characters in the JIS X0208 standard include Japanese syllabic characters (Hiragana and Katakana), ideographic characters (Kanji), Roman, Greek, and Cyrillic alphabets, punctuation marks, and miscellaneous symbols. For example, the JIS X0208 standard indexes the 83 Hiragana as `0x2421`...`0x2473`. To obtain the EUC code for any JIS X0208 character, you would add `0x80` to each byte (i.e., 'set the high bit' on each byte). So the first of the 83 Hiragana (`0x2421`) would be encoded as the successive pair of bytes `0xa4` and `0xa1`.

The implementation of the JIS X0208 standard in the HersheyEUC font is based on Dr. Hershey's digitizations, and is complete enough to be useful. All 83 Hiragana and 86 Katakana are available, though the little-used 'half-width Katakana' are not supported. Also, 603 Kanji are available, including 596 of the 2965 JIS Level 1 (i.e., frequently used) Kanji. The Hiragana, the Katakana, and the available Kanji all have the same width. The file '`kanji.doc`', which on most systems is installed in '`/usr/share/libplot`' or '`/usr/local/share/libplot`', lists the 603 available Kanji. Each JIS X0208 character that is unavailable will be drawn as an 'undefined character' glyph (a bundle of horizontal lines).

The eight Hewlett–Packard vector fonts in the ArcANK and StickANK typefaces are also used for displaying Japanese text. They are available when producing HP-GL output for the HP7550A graphics plotter and the HP758x, HP7595A and HP7596A drafting plotters. To ensure that they are available, you must set the environment variable or driver parameter `HPGL_VERSION` to "1.5".

ANK stands for Alphabet, Numerals, and Katakana. The ANK fonts use the 'Kana-8' encoding. The lower half of each font uses the JIS Roman encoding, and the upper half contains half-width Katakana. Half-width Katakana are simplified Katakana that may need to be equipped with diacritical marks. The diacritical marks are included in the encoding, as separate characters.

A.3 Available text fonts for the X Window System

The plotting utilities `graph -T X`, `plot -T X`, `tek2plot -T X`, `pic2plot -T X`, and `plotfont -T X`, and the `libplot` library that they are built on, can draw text on an X Window System display in a wide variety of fonts. This includes the 22 built-in Hershey vector fonts. They can use the 35 built-in Postscript fonts too, if those fonts are available on the X display. Most releases of the plotting utilities include freely distributable versions of the 35 Postscript fonts, in Type 1 format, that are easily installed on any X display.

In fact, the plotting utilities can use most fonts that are available on the current X display. This includes all scalable fonts that have a so-called XLFD (X Logical Font Description) name. For example, the "CharterBT-Roman" font is available on many X displays. It has a formal XLFD name, namely "-bitstream-charter-medium-r-normal–0-0-0-0-p-0-iso8859-1". The plotting utilities would refer to it as "charter-medium-r-normal". The command

```
echo 0 0 1 1 2 0 | graph -T X -F charter-medium-r-normal
```

would draw a plot in a popped-up X window, in which all axis ticks are labeled in this font.

You may determine which fonts are available on an X display by using the `xlsfonts` command. Fonts whose names end in "-0-0-0-0-p-0-iso8859-1" or "-0-0-0-0-m-0-iso8859-1" are scalable ISO-Latin-1 fonts that can be used by `libplot`'s X Plotters and by the plotting utilities that are built on `libplot`. The two sorts of font are variable-width and fixed-width fonts, respectively. Fonts whose names end in "iso8859-2", etc., and "adobe-fontspecific", may also be used, even though they do not employ the standard ISO-Latin-1 encoding.

The escape sequences that provide access to the non-ASCII '8-bit' characters in the built-in ISO-Latin-1 fonts may be employed when using any ISO-Latin-1 X Window System font. For more on escape sequences, see Section A.4 [Text String Format], page 131. As an example, "\Po" will yield the British pounds sterling symbol '£'. The command

```
echo 0 0 1 1 | graph -T X -F charter-medium-r-normal -L "A \Po1 Plot"
```

shows how this symbol could be used in a graph label. In the same way, the escape sequences that provide access to mathematical symbols and Greek characters may be employed when using any X Window System font, whether or not it is an ISO-Latin-1 font.

The plotting utilities, including `graph`, support a `--bitmap-size` option. It is meaningful only if the '`-T X`' option is used, since it sets the size of the popped-up X Window. You may use it to obtain some interesting visual effects. Each of the plotting utilities assumes that it is drawing in a square region, so if you use the '`--bitmap-size 800x400`' option, your plot will be scaled anisotropically, by a larger factor in the horizontal direction than in the vertical direction. The fonts in the plot will be scaled in the same way. Actually, this requires a modern (X11R6) display. If your X display cannot scale a font, a default scalable font (such as "HersheySerif") will be substituted.

A.4 Text string format and escape sequences

Text strings that are drawn by `libplot`, and by such applications as `graph`, `plot`, `tek2plot`, `pic2plot`, and `plotfont`, which are built on `libplot`, must consist of printable characters. No embedded control characters, such as newlines or carriage returns, are allowed. Technically, a character is 'printable' if it comes from either of the two byte ranges

`0x20...0x7e` and `0xa0...0xff`. The former is the printable ASCII range and the latter is the printable '8-bit' range.

Text strings may, however, include embedded 'escape sequences' that shift the font, append subscripts or superscripts, or include non-ASCII characters and mathematical symbols. As a consequence, the axis labels on a plot prepared with `graph` may include such features. So may the text strings that `pic2plot` uses to label objects.

The format of the escape sequences should look familiar to anyone who is familiar with the TEX or `groff` document formatters. Each escape sequence consists of three characters: a backslash and two additional characters. The most frequently used escape sequences are as follows.

"\sp" start superscript mode

"\ep" end superscript mode

"\sb" start subscript mode

"\eb" end subscript mode

"\mk" mark location

"\rt" return to marked location

For example, the string "x\sp2\ep" would be interpreted as 'x squared'. Subscripts on subscripts, etc., are allowed. Subscripts and superscripts may be vertically aligned by judicious use of the "\mk" and "\rt" escape sequences. For example, "a\mk\sbi\eb\rt\sp2\ep" produces "a sub i squared", with the exponent '2' placed immediately above the subscript.

There are also escape sequences that switch from font to font within a typeface. For an enumeration of the fonts within each typeface, see Section A.1 [Text Fonts], page 124. Suppose for example that the current font is Times-Roman, which is font #1 in the 'Times' typeface. The string "A \f2very\f1 well labeled axis" would be a string in which the word 'very' appears in Times-Italic rather than Times-Roman. That is because Times-Italic is the #2 font in the typeface. Font-switching escape sequences are of the form "\f*n*", where *n* is the number of the font to be switched to. For compatibility with `groff`, "\fR", "\fI", "\fB" are equivalent to "\f1", "\f2", "\f3", respectively. "\fP" will switch the font to the previously used font (only one font is remembered). There is currently no support for switching between fonts in different typefaces.

There are also a few escape sequences for horizontal shifts, which are useful for improving horizontal alignment, such as when shifting between italic and non-italic fonts. "\r1", "\r2", "\r4", "\r6", "\r8", and "\r^" are escape sequences that shift right by 1 em, 1/2 em, 1/4 em, 1/6 em, 1/8 em, and 1/12 em, respectively. "\l1", "\l2", "\l4", "\l6", "\l8", and "\l^" are similar, but shift left instead of right. "A \fIvery\r^\fP well labeled axis" would look slightly better than "A \fIvery\fP well labeled axis".

Square roots are handled with the aid of a special pair of escape sequences, together with the "\mk" and "\rt" sequences discussed above. A square root symbol is begun with "\sr", and continued arbitrarily far to the right with the overbar ('run') escape sequence, "\rn". For example, the string "\sr\mk\rn\rn\rtab" would be plotted as 'the square root of ab'. To adjust the length of the overbar, you may need to experiment with the number of times "\rn" appears.

To underline a string, you would use `"\ul"`, the underline escape sequence, one or more times. The `"\mk"`...`"\rt"` trick would be employed in the same way. So, for example, `"\mk\ul\ul\ul\rtabc"` would yield an underlined `"abc"`. To adjust the length of the underline, you may need to experiment with the number of times `"\ul"` appears. You may also need to use one or more of the abovementioned horizontal shifts. For example, if the "HersheySerif" font were used, `"\mk\ul\ul\l8\ul\rtabc"` would yield a better underline than `"\mk\ul\ul\ul\rtabc"`.

Besides the preceding escape sequences, there are also escape sequences for the printable non-ASCII characters in each of the built-in ISO-Latin-1 fonts (which means in every built-in font, except for the symbol fonts, the HersheyCyrillic fonts, HersheyEUC, and ZapfDingbats). The useful non-ASCII characters include accented characters among others. Such '8-bit' characters, in the `0xa0`...`0xff` byte range, may be included directly in a text string. But if your terminal does not permit this, you may use the escape sequences for them instead.

There are escape sequences for the mathematical symbols and Greek characters in the symbol fonts, as well. This is how the symbol fonts are usually accessed. Which symbol font the mathematical symbols and Greek characters are taken from depends on whether your current font is a Hershey font or a non-Hershey font. They are taken from the HersheySerifSymbol font or the HersheySansSymbol font in the former case, and from the Symbol font in the latter.

The following are the escape sequences that provide access to the non-ASCII characters of the current font, provided that it is an ISO-Latin-1 font. Each escape sequence is followed by the position of the corresponding character in the ISO-Latin-1 encoding (in decimal), and the official Postscript name of the character. Most names should be self-explanatory. For example, 'eacute' is a lower-case 'e', equipped with an acute accent.

`"\r!"` [161] exclamdown

`"\ct"` [162] cent

`"\Po"` [163] sterling

`"\Cs"` [164] currency

`"\Ye"` [165] yen

`"\bb"` [166] brokenbar

`"\sc"` [167] section

`"\ad"` [168] dieresis

`"\co"` [169] copyright

`"\Of"` [170] ordfeminine

`"\Fo"` [171] guillemotleft

`"\no"` [172] logicalnot

`"\hy"` [173] hyphen

`"\rg"` [174] registered

"\a-"	[175] macron
"\de"	[176] degree
"\+-"	[177] plusminus
"\S2"	[178] twosuperior
"\S3"	[179] threesuperior
"\aa"	[180] acute
"*m"	[181] mu
"\ps"	[182] paragraph
"\md"	[183] periodcentered
"\ac"	[184] cedilla
"\S1"	[185] onesuperior
"\Om"	[186] ordmasculine
"\Fc"	[187] guillemotright
"\14"	[188] onequarter
"\12"	[189] onehalf
"\34"	[190] threequarters
"\r?"	[191] questiondown
"\‘A"	[192] Agrave
"\’A"	[193] Aacute
"\^A"	[194] Acircumflex
"\~A"	[195] Atilde
"\:A"	[196] Adieresis
"\oA"	[197] Aring
"\AE"	[198] AE
"\,C"	[199] Ccedilla
"\‘E"	[200] Egrave
"\’E"	[201] Eacute
"\^E"	[202] Ecircumflex
"\:E"	[203] Edieresis
"\‘I"	[204] Igrave
"\’I"	[205] Iacute
"\^I"	[206] Icircumflex

"\:I"	[207] Idieresis
"\-D"	[208] Eth
"\~N"	[209] Ntilde
"\’O"	[210] Ograve
"\’O"	[211] Oacute
"\^O"	[212] Ocircumflex
"\~O"	[213] Otilde
"\:O"	[214] Odieresis
"\mu"	[215] multiply
"\/O"	[216] Oslash
"\‘U"	[217] Ugrave
"\’U"	[218] Uacute
"\^U"	[219] Ucircumflex
"\:U"	[220] Udieresis
"\’Y"	[221] Yacute
"\TP"	[222] Thorn
"\ss"	[223] germandbls
"\‘a"	[224] agrave
"\’a"	[225] aacute
"\^a"	[226] acircumflex
"\~a"	[227] atilde
"\:a"	[228] adieresis
"\oa"	[229] aring
"\ae"	[230] ae
"\,c"	[231] ccedilla
"\‘e"	[232] egrave
"\’e"	[233] eacute
"\^e"	[234] ecircumflex
"\:e"	[235] edieresis
"\‘i"	[236] igrave
"\’i"	[237] iacute
"\^i"	[238] icircumflex

"\:i" [239] idieresis

"\Sd" [240] eth

"\~n" [241] ntilde

"\`o" [242] ograve

"\'o" [243] oacute

"\^o" [244] ocircumflex

"\~o" [245] otilde

"\:o" [246] odieresis

"\di" [247] divide

"\/o" [248] oslash

"\`u" [249] ugrave

"\'u" [250] uacute

"\^u" [251] ucircumflex

"\:u" [252] udieresis

"\'y" [253] yacute

"\Tp" [254] thorn

"\:y" [255] ydieresis

The following are the escape sequences that provide access to mathematical symbols and Greek characters in the current symbol font, whether HersheySerifSymbol or Hershey-SansSymbol (for Hershey fonts) or Symbol (for Postscript fonts). Each escape sequence is followed by the position (in octal) of the corresponding character in the symbol encoding, and the official Postscript name of the character. Many escape sequences and names should be self-explanatory. "*a" represents a lower-case Greek alpha, for example. For a table displaying each of the characters below, see the *Postscript Language Reference Manual*.

"\fa" [0042] universal

"\te" [0044] existential

"\st" [0047] suchthat

"**" [0052] asteriskmath

"\=~" [0100] congruent

"*A" [0101] Alpha

"*B" [0102] Beta

"*X" [0103] Chi

"*D" [0104] Delta

"*E" [0105] Epsilon

"*F"	[0106] Phi
"*G"	[0107] Gamma
"*Y"	[0110] Eta
"*I"	[0111] Iota
"\+h"	[0112] theta1
"*K"	[0113] Kappa
"*L"	[0114] Lambda
"*M"	[0115] Mu
"*N"	[0116] Nu
"*O"	[0117] Omicron
"*P"	[0120] Pi
"*H"	[0121] Theta
"*R"	[0122] Rho
"*S"	[0123] Sigma
"*T"	[0124] Tau
"*U"	[0125] Upsilon
"\ts"	[0126] sigma1
"*W"	[0127] Omega
"*C"	[0130] Xi
"*Q"	[0131] Psi
"*Z"	[0132] Zeta
"\tf"	[0134] therefore
"\pp"	[0136] perpendicular
"\ul"	[0137] underline
"\rx"	[0140] radicalex
"*a"	[0141] alpha
"*b"	[0142] beta
"*x"	[0143] chi
"*d"	[0144] delta
"*e"	[0145] epsilon
"*f"	[0146] phi
"*g"	[0147] gamma

"*y"	[0150] eta
"*i"	[0151] iota
"\+f"	[0152] phi1
"*k"	[0153] kappa
"*l"	[0154] lambda
"*m"	[0155] mu
"*n"	[0156] nu
"*o"	[0157] omicron
"*p"	[0160] pi
"*h"	[0161] theta
"*r"	[0162] rho
"*s"	[0163] sigma
"*t"	[0164] tau
"*u"	[0165] upsilon
"\+p"	[0166] omega1
"*w"	[0167] omega
"*c"	[0170] xi
"*q"	[0171] psi
"*z"	[0172] zeta
"\ap"	[0176] similar
"\+U"	[0241] Upsilon1
"\fm"	[0242] minute
"\<="	[0243] lessequal
"\f/"	[0244] fraction
"\if"	[0245] infinity
"\Fn"	[0246] florin
"\CL"	[0247] club
"\DI"	[0250] diamond
"\HE"	[0251] heart
"\SP"	[0252] spade
"\<>"	[0253] arrowboth
"\<-"	[0254] arrowleft

"\ua"	[0255] arrowup
"\->"	[0256] arrowright
"\da"	[0257] arrowdown
"\de"	[0260] degree
"\+-"	[0261] plusminus
"\sd"	[0262] second
"\>="	[0263] greaterequal
"\mu"	[0264] multiply
"\pt"	[0265] proportional
"\pd"	[0266] partialdiff
"\bu"	[0267] bullet
"\di"	[0270] divide
"\!="	[0271] notequal
"\=="	[0272] equivalence
"\~~"	[0273] approxequal
"\.."	[0274] ellipsis
NONE	[0275] arrowvertex
"\an"	[0276] arrowhorizex
"\CR"	[0277] carriagereturn
"\Ah"	[0300] aleph
"\Im"	[0301] Ifraktur
"\Re"	[0302] Rfraktur
"\wp"	[0303] weierstrass
"\c*"	[0304] circlemultiply
"\c+"	[0305] circleplus
"\es"	[0306] emptyset
"\ca"	[0307] cap
"\cu"	[0310] cup
"\SS"	[0311] superset
"\ip"	[0312] reflexsuperset
"\n<"	[0313] notsubset
"\SB"	[0314] subset

"\ib"	[0315]	reflexsubset
"\mo"	[0316]	element
"\nm"	[0317]	notelement
"\/_"	[0320]	angle
"\gr"	[0321]	nabla
"\rg"	[0322]	registerserif
"\co"	[0323]	copyrightserif
"\tm"	[0324]	trademarkserif
"\PR"	[0325]	product
"\sr"	[0326]	radical
"\md"	[0327]	dotmath
"\no"	[0330]	logicalnot
"\AN"	[0331]	logicaland
"\OR"	[0332]	logicalor
"\hA"	[0333]	arrowdblboth
"\lA"	[0334]	arrowdblleft
"\uA"	[0335]	arrowdblup
"\rA"	[0336]	arrowdblright
"\dA"	[0337]	arrowdbldown
"\lz"	[0340]	lozenge
"\la"	[0341]	angleleft
"\RG"	[0342]	registersans
"\CO"	[0343]	copyrightsans
"\TM"	[0344]	trademarksans
"\SU"	[0345]	summation
NONE	[0346]	parenlefttp
NONE	[0347]	parenleftex
NONE	[0350]	parenleftbt
"\lc"	[0351]	bracketlefttp
NONE	[0352]	bracketleftex
"\lf"	[0353]	bracketleftbt
"\lt"	[0354]	bracelefttp

"\lk" [0355] braceleftmid

"\lb" [0356] braceleftbt

"\bv" [0357] braceex

"\eu" [0360] euro

"\ra" [0361] angleright

"\is" [0362] integral

NONE [0363] integraltp

NONE [0364] integralex

NONE [0365] integralbt

NONE [0366] parenrighttp

NONE [0367] parenrightex

NONE [0370] parenrightbt

"\rc" [0371] bracketrighttp

NONE [0372] bracketrightex

"\rf" [0373] bracketrightbt

"\RT" [0374] bracerighttp

"\rk" [0375] bracerightmid

"\rb" [0376] bracerightbt

Finally, there are escape sequences that apply only if the current font is a Hershey font. Most of these escape sequences provide access to special symbols that belong to no font, and are accessible by no other means. These symbols are of two sorts: miscellaneous, and astronomical or zodiacal. The escape sequences for the miscellaneous symbols are as follows.

"\dd" daggerdbl

"\dg" dagger

"\hb" hbar

"\li" lineintegral

"\IB" interbang

"\Lb" lambdabar

"\~-" modifiedcongruent

"\-+" minusplus

"\||" parallel

"\s-" [variant form of s]

The final escape sequence in the table above, "\s-", yields a letter rather than a symbol. It is provided because in some Hershey fonts, the shape of the lower-case letter 's' differs if it is the last letter in a word. This is the case for HersheyGothicGerman. The German word "besonders", for example, should be written as "besonder\s-" if it is to be rendered correctly in this font. The same is true for the two Hershey symbol fonts, with their Greek alphabets (in Greek text, lower-case final 's' is different from lower-case non-final 's'). In Hershey fonts where there is no distinction between final and non-final 's', "s" and "\s-" are equivalent.

The escape sequences for the astronomical symbols, including the signs for the twelve constellations of the zodiac, are listed in the following table. We stress that that like the preceding miscellaneous escape sequences, they apply only if the current font is a Hershey font.

"\SO"	sun
"\ME"	mercury
"\VE"	venus
"\EA"	earth
"\MA"	mars
"\JU"	jupiter
"\SA"	saturn
"\UR"	uranus
"\NE"	neptune
"\PL"	pluto
"\LU"	moon
"\CT"	comet
"\ST"	star
"\AS"	ascendingnode
"\DE"	descendingnode
"\AR"	aries
"\TA"	taurus
"\GE"	gemini
"\CA"	cancer
"\LE"	leo
"\VI"	virgo
"\LI"	libra
"\SC"	scorpio
"\SG"	sagittarius

"\CP" capricornus

"\AQ" aquarius

"\PI" pisces

The preceding miscellaneous and astronomical symbols are not the only special non-font symbols that can be used if the current font is a Hershey font. The entire library of glyphs digitized by Allen Hershey is built into GNU `libplot`. So text strings may include any Hershey glyph. Each of the available Hershey glyphs is identified by a four-digit number. Standard Hershey glyph #1 would be specified as "\#H0001". The standard Hershey glyphs range from "\#H0001" to "\#H3999", with a number of gaps. Some additional glyphs designed by others appear in the "\#H4000"..."\#H4194" range. Syllabic Japanese characters (Kana) are located in the "\#H4195"..."\#H4399" range.

You may order a table of nearly all the Hershey glyphs in the "\#H0001"..."\#H3999" range from the U.S. National Technical Information Service, at +1 703 487 4650. Ask for item number PB251845; the current price is about US$40. By way of example, the string

```
"\#H0744\#H0745\#H0001\#H0002\#H0003\#H0869\#H0907\#H2330\#H2331"
```

when drawn will display a shamrock, a fleur-de-lys, cartographic (small) letters A, B, C, a bell, a large circle, a treble clef, and a bass clef. Again, this assumes that the current font is a Hershey font.

You may also use Japanese syllabic characters (Hiragana and Katakana) and ideographic characters (Kanji) when drawing strings in any Hershey font. In all, 603 Kanji are available; these are the same Kanji that are available in the HersheyEUC font. The Japanese characters are indexed according to the JIS X0208 standard for Japanese typography, which represents each character by a two-byte sequence. The file '`kanji.doc`', which is distributed along with the GNU plotting utilities, lists the available Kanji. On most systems it is installed in '`/usr/share/libplot`' or '`/usr/local/share/libplot`'.

Each JIS X0208 character would be specified by an escape sequence which expresses this two-byte sequence as four hexadecimal digits, such as "\#J357e". Both bytes must be in the `0x21`...`0x7e` range in order to define a JIS X0208 character. Kanji are located at "\#J3021" and above. Characters appearing elsewhere in the JIS X0208 encoding may be accessed similarly. For example, Hiragana and Katakana are located in the "\#J2421"..."\#J257e" range, and Roman characters in the "\#J2321"..."\#J237e" range. The file '`kana.doc`', which is installed in the same directory as '`kanji.doc`', lists the encodings of the Hiragana and Katakana. For more on the JIS X0208 standard, see Ken Lunde's *Understanding Japanese Information Processing* (O'Reilly, 1993), and http://www.ora.com/people/authors/lunde/cjk_inf.html, his on-line supplement.

The Kanji numbering used in A. N. Nelson's *Modern Reader's Japanese-English Character Dictionary*, a longtime standard, is also supported. (This dictionary is published by C. E. Tuttle and Co., with ISBN 0-8048-0408-7. A revised edition [ISBN 0-8048-2036-8] appeared in 1997, but uses a different numbering.) 'Nelson' escape sequences for Kanji are similar to JIS X0208 escape sequences, but use four decimal instead of four hexadecimal digits. The file '`kanji.doc`' gives the correspondence between the JIS numbering scheme and the Nelson numbering scheme. For example, "\#N0001" is equivalent to "\#J306c". It also gives the positions of the available Kanji in the Unicode encoding.

All available Kanji have the same width, which is the same as that of the syllabic Japanese characters (Hiragana and Katakana). Each Kanji that is not available will print as an 'undefined character' glyph (a bundle of horizontal lines). The same is true for non-Kanji JIS X0208 characters that are not available.

A.5 Available marker symbols

The GNU `libplot` library supports a standard set of marker symbols, numbered 0 through 31. These are the symbols that the `graph` program will plot at each point of a dataset, if the '`-S`' option is used. The list is as follows (by convention, marker symbol #0 means no symbol at all).

1. dot ($\cdot$)
2. plus ($+$)
3. asterisk ($*$)
4. circle ($\circ$)
5. cross ($\times$)
6. square
7. triangle
8. diamond
9. star
10. inverted triangle
11. starburst
12. fancy plus
13. fancy cross
14. fancy square
15. fancy diamond
16. filled circle
17. filled square
18. filled triangle
19. filled diamond
20. filled inverted triangle
21. filled fancy square
22. filled fancy diamond
23. half filled circle
24. half filled square
25. half filled triangle
26. half filled diamond
27. half filled inverted triangle
28. half filled fancy square
29. half filled fancy diamond

30. octagon
31. filled octagon

The interpretation of marker symbols 1 through 5 is the same as in the well known GKS (Graphical Kernel System).

Symbols 32 and up are interpreted as characters in a certain text font. For `libplot`, it is the current font. For `graph`, it is the font selected with the '`--symbol-font-name`' option. By default, this is the ZapfDingbats font except in `graph -T pnm`, `graph -T gif`, `graph -T pcl`, `graph -T hpgl` and `graph -T tek`. These variants of `graph` normally have no access to Postscript fonts, so they use the HersheySerif font instead.

Many of the characters in the ZapfDingbats font are suitable for use as marker symbols. For example, character #74 is the Texas star. Doing

```
echo 0 0 1 2 2 1 3 2 4 0 | graph -T ps -m 0 -S 74 0.1 > plot.ps
```

will produce a Postscript plot consisting of five data points, not joined by line segments. Each data point will be marked by a Texas star, of a large font size (0.1 times the width of the plotting box).

If you are using `graph -T pcl` or `graph -T hpgl` and wish to use font characters as marker symbols, you should consider using the Wingdings font, which is available when producing PCL 5 or HP-GL/2 output. Doing

```
echo 0 0 1 2 2 1 3 2 4 0 |
    graph -T pcl -m 0 --symbol-font Wingdings -S 181 0.1 > plot.pcl
```

will produce a PCL 5 plot that is similar to the preceding Postscript plot. The Wingdings font has the Texas star in location #181.

Appendix B Specifying Colors by Name

Many of the plotting utilities allow colors to be specified by name. For example, `graph` supports the '`--frame-color`' option. The other graphics programs support the '`--pen-color`' option, and they all, including `graph` support the '`--bg-color`' option. The `libplot` library, on which the graphics programs are based, includes the `pencolorname`, `fillcolorname`, and `bgcolorname` functions.

In any of these contexts, 665 distinct color names are recognized, including obscure ones like "dark magenta", "forest green", and "olive drab". Color names are case-insensitive, and spaces are ignored. So, for example, "RosyBrown" is equivalent to "rosy brown", and "DarkGoldenrod3" to "dark goldenrod 3".

The file '`colors.txt`', which is distributed along with the GNU plotting utilities, lists the available color names. On most systems it is installed in '`/usr/share/libplot`' or '`/usr/local/share/libplot`'. The color names are essentially those recognized by recent releases of the X Window System, which on most machines are listed in the file '`/usr/lib/X11/rgb.txt`'. However, for every color name containing the string "gray", a version containing "grey" has been included. For example, both "dark slate gray 4" and "dark slate grey 4" are recognized color names.

Appendix C The Graphics Metafile Format

GNU graphics metafiles are produced by the raw variants of `graph`, `plot`, `pic2plot`, `tek2plot`, and `plotfont`, and by any other graphics application that uses the Metafile Plotter support contained in GNU `libplot`. A file in this format is a sort of audit trail: a sequence of plotting commands, each of which may be followed by data. Each plotting command is an 'op code': a single ASCII character, indicating a Plotter operation. The data following the command are the arguments passed to the operation, if any.

There are two sorts of GNU graphics metafile: binary (the default) and portable (human-readable). Binary metafiles begin with the magic string "#PLOT 1\n", and portable metafiles with the magic string "#PLOT 2\n". If you wish to transfer metafiles between machines of different types, you should use portable rather than binary format. Portable metafiles are produced by GNU `graph` and the other plotting utilities if the '`-O`' option is specified, and by Metafile Plotters if the `META_PORTABLE` parameter is set to "yes". Both binary and portable metafiles can be translated to other formats by GNU `plot`.

In the portable format, the arguments of each operation (integers, floating point numbers, or strings) are printed in a human-readable form, separated by spaces, and each argument list ends with a newline. In the binary format, the arguments are represented as integers, single precision floating point numbers, or newline-terminated ASCII strings. Using the newline character as a terminator is acceptable because each `libplot` operation includes a maximum of one string among its arguments, and such a string may not include a newline. Also, the string must come last among the arguments.

The `openpl` and `closepl` operations open and close a Plotter, i.e., begin and end a page of graphics. They are represented by the op codes '`o`' and '`x`', respectively. The `erase` operation, if present, separates frames within a page. On real-time display devices, it is interpreted as a screen erasure. It is represented by the op code '`e`'.

Each of the 89 other Plotter operations has a corresponding op code, with 12 exceptions. These 12 exceptions are (1) the setup operation `flushpl`, (2) the operations `havecap`, `labelwidth`, and `flabelwidth`, which merely return information, (3) the `colorname`, `pencolorname`, `fillcolorname`, and `bgcolorname` operations, which are internally mapped to `pencolor`, `fillcolor`, and `bgcolor`, (4) the `frotate`, `fscale`, and `ftranslate` operations, which are internally mapped to `fconcat`, and (5) the `ffontname` operation, which in a metafile would be indistinguishable from `fontname`. So besides '`o`' and '`x`', there are 78 possible op codes, for a total of 80. The following table lists 10 of the op codes other than '`o`' and '`x`', followed by the name of the `libplot` operation they stand for.

Op Code	Operation
'`a`'	`arc`
'`c`'	`circle`
'`e`'	`erase`
'`f`'	`linemod`
'`l`'	`line`
'`m`'	`move`

'n' `cont`

'p' `point`

's' `space`

't' `label`

The full set of 80 op codes is listed in the header file '`plot.h`', which is distributed along with the plotting utilities. On most systems it is installed in '`/usr/include`' or '`/usr/local/include`'.

It is worth noting that of the 80 op codes, only 50 are used in portable metafiles. That is because in ASCII format, there is no point in distinguishing the floating point `libplot` operations from their integer counterparts.

The 10 op codes in the table above are actually the op codes of the traditional 'plot(5)' format produced by pre-GNU versions of `graph` and `libplot`. The use of these op codes make GNU metafile format compatible with plot(5) format. The absence of a magic string, and of the '`o`' and '`x`' op codes, makes it possible to distinguish files in plot(5) format from GNU metafiles. GNU `plot` can convert files in plot(5) format to GNU metafiles in either binary or portable format.

Appendix D Obtaining Auxiliary Software

D.1 How to get idraw

The `idraw` utility mentioned several times in this documentation is a freely distributable interactive drawing editor for the X Window System. It may be used to edit the output of `graph -T ps`, or, in general, the output of any application that uses the Postscript Plotter support contained in `libplot`.

The current version of `idraw` is maintained by Vectaport, Inc., and is available at http://www.vectaport.com, their Web site. It is part of the `ivtools` package, which is a framework for building custom drawing editors. `idraw` was originally part of the `InterViews` package, developed by Stanford University and Silicon Graphics. The `InterViews` package is available at ftp://interviews.stanford.edu, a distribution site but is no longer supported. Retrieving the `ivtools` package instead is recommended.

Also available at http://www.vectaport.com, Vectaport's Web site is an enhanced version of `idraw` called `drawtool`. `drawtool` can import bitmapped graphics in TIFF and PBM/PGM/PPM formats, as well as in the X11 bitmap format that `idraw` can import.

D.2 How to get xfig

The `xfig` utility mentioned several times in this documentation is a freely distributable interactive drawing editor for the X Window System. It may be used to edit the output of `graph -T fig`, or, in general the output of any application that uses the Fig Plotter support contained in `libplot`.

The current version is available at ftp://ftp.x.org/contrib/applications/drawing_tools/. It can import graphics in GIF, X11 bitmap, and Postscript formats. Accompanying the editor is a package called `transfig`, which allows `xfig` graphics to be exported in many formats. GIF, X11 bitmap, LaTeX, and Postscript formats are supported.

There is http://duke.usask.ca/~macphed/soft/fig, a Web page on Fig format, which discusses application software packages that can interoperate with `xfig`.

Acknowledgements

Several of the GNU plotting utilities were inspired by Unix plotting utilities. A `graph` utility and various plot filters were present in the first releases of Unix from Bell Laboratories, going at least as far back as the Version 4 distribution (1973). The first supported display device was a Tektronix 611 storage scope. Most of the work on tying the plot filters together and breaking out device-dependent versions of `libplot` was performed by ⟨`llc@research.att.com, Lorinda Cherry`⟩. By the time of Version 7 Unix (1979) and the subsequent Berkeley releases, the package consisting of `graph`, `plot`, `spline`, and several device-dependent versions of `libplot` was a standard Unix feature. Supported devices by the early 1980's included Tektronix storage scopes, early graphics terminals, 200 dpi electrostatic printer/plotters from Versatec and Varian, and pen plotters from Hewlett–Packard.

In 1989, ⟨`rich@freebsd.org, Rich Murphey`⟩ wrote the first GNU versions of `graph`, `plot`, and `spline`, and the earliest documentation. Richard Stallman further directed development of the programs and provided editorial support for the documentation. ⟨`interran@uluru.stanford.edu, John Interrante`⟩, of the InterViews team at Stanford, generously provided the `idraw` Postscript prologue now included in `libplot`, and helpful comments. The package as it stood in 1991 was distributed under the name 'GNU graphics'.

In 1995 ⟨`rsm@math.arizona.edu, Robert Maier`⟩ took over development of the package, and designed and wrote the current, maximally device-independent, standalone version of `libplot`. He also rewrote `graph` from scratch, turning it into a real-time filter that would use the new library. He fleshed out `spline` too, by adding support for splines in tension, periodicity, and cubic Bessel interpolation.

`libplot` now incorporates the X Window System code for filling polygons and drawing wide polygonal lines and arcs. This code is used when producing output in bitmap formats (PNM and pseudo-GIF). It was written by Brian Kelleher, Joel McCormack, Todd Newman, Keith Packard, Robert Scheifler and Ken Whaley, who worked for Digital Equipment Corp., MIT, and/or the X Consortium, and is copyright © 1985–89 by the X Consortium.

The pseudo-GIF support now in `libplot` uses the 'miGIF' run-length encoding routines developed by ⟨`mouse@rodents.montreal.qc.ca, der Maus`⟩ and ⟨`ivo@hasc.com, ivo`⟩, which are copyright © 1998 by http://www.hasc.com, Hutchison Avenue Software Corporation. The copyright notice and permission notice for the miGIF routines are distributed with the source code distribution of the plotting utilities.

Most development work on `ode` was performed by ⟨`nbt@reed.edu, Nick Tufillaro`⟩ in 1978–1994, on a sequence of platforms that extended back to a PDP-11 running Version 4 Unix. In 1997 Robert modified Nick's 1994 version to agree with GNU conventions on coding and command-line parsing, extended it to support the full set of special functions supported by `gnuplot`, and extended the exception handling.

Many other people aided the development of the plotting utilities package along the way. The Hershey vector fonts now in `libplot` are of course based on the characters digitized in the mid to late 1960's by Allen V. Hershey, who deserves a vote of thanks. Additional characters and/or marker symbols were taken from the SLAC Unified Graphics System developed by Robert C. Beach in the mid-1970's, and from the fonts designed by ⟨`wolff@inf.fu-berlin.de, Thomas Wolff`⟩ for Ghostscript. The interpolation algorithms used in `spline` are based on the algorithms of ⟨`cline@cs.utexas.edu, Alan K. Cline`⟩, as described in his papers in the Apr. 1974 issue of *Communications of the ACM*. The table-driven parser used in `tek2plot` was written at Berkeley in the mid-1980's by ⟨`moy@parc.xerox.com, Edward Moy`⟩. The 'sagitta' algorithm used in an extended form in `libplot` for drawing circular and elliptic arcs was developed by Peter Karnow of URW and ⟨`turk@apple.com, Ken Turkowski`⟩ of Apple. ⟨`toy@rtp.ericsson.se, Raymond Toy`⟩ helped with the tick mark spacing code in `graph` and was the first to incorporate GNU `getopt`. Arthur Smith, formerly of LASSP at Cornell, provided code for his `xplot` utility. ⟨`beebe@math.utah.edu, Nelson Beebe`⟩ exhaustively tested the package installation process.

Robert Maier wrote the documentation, which now incorporates Nick Tufillaro's `ode` manual. Julie Sussmann checked over the documentation for style and clarity.

Table of Contents