The PLplot Plotting Library
Programmer's Reference Manual
Version 5.0

Maurice J. LeBrun
Geoff Furnish

Institute for Fusion Studies
University of Texas at Austin

September 6, 1994

Contents

Part I

Introduction

Chapter 1

Introduction

This chapter introduces the incredibly cool PLplot scientific plotting package.

1.1 The PLplot Plotting Library

PLplot is a library of C functions that are useful for making scientific plots from a program written in C, C++, or Fortran. The PLplot library can be used to create standard x-y plots, semilog plots, log-log plots, contour plots, 3D plots, mesh plots, bar charts and pie charts. Multiple graphs (of the same or different sizes) may be placed on a single page with multiple lines in each graph. Different line styles, widths and colors are supported. A virtually infinite number of distinct area fill patterns may be used. There are almost 1000 characters in the extended character set. This includes four different fonts, the Greek alphabet and a host of mathematical, musical, and other symbols. The fonts can be scaled to any desired size. A variety of output devices are supported and new devices can be easily added by writing a small number of device dependent routines.

PLplot was originally developed by Sze Tan of the University of Auckland in Fortran-77. Many of the underlying concepts used in the PLplot package are based on ideas used in Tim Pearson's PGPLOT package. Sze Tan writes:

> I'm rather amazed how far PLPLOT has travelled given its origins etc. I first used PGPLOT on the Starlink VAX computers while I was a graduate student at the Mullard Radio Astronomy Observatory in Cambridge from 1983-1987. At the beginning of 1986, I was to give a seminar within the department at which I wanted to have a computer graphics demonstration on an IBM PC which was connected to a completely non-standard graphics card. Having about a week to do this and not having any drivers for the card, I started from the back end and designed PLPLOT to be such that one only needed to be able to draw a line or a dot on the screen in order to do arbitrary graphics. The application programmer's interface was made as similar as possible to PGPLOT so that I could easily port my programs from the VAX to the PC. The kernel of PLPLOT was modelled on PGPLOT but the code is not derived from it.

The C version of PLplot was developed by Tony Richardson on a Commodore Amiga. In the process, several of the routines were rewritten to improve efficiency and some new features added. The program structure was changed somewhat to make it easier to incorporate new devices. Additional features were added to allow three-dimensional plotting and better access to low-level routines.

PLplot 5.0 is a continuation of our work on PLplot 4.0, which never got widely distributed. It became clear during the work on 4.0 that in order to support an interactive driver under Unix (using Tcl/Tk), many additions to the basic capabilities of the package were needed. So without stopping to fully document and bug-fix the 4.0 additions, work on 5.0 was begun. The result is that a very capable PLplot-based widget for the Tk toolkit has been written. This widget can manipulate the plot (zoom/pan, scale, orient, change colors), as well dump it to any supported device. There are help menus and user customization options. These are still in the process of being documented.

Other changes include the introduction of a new color palette (cmap1) for smooth color shaded images (typically for 2d or 3d plots – in which color represents function intensity), support for color fill plots, and lots more cool stuff. The manual has been rewritten in LaTeXinfo, so that there is now a printed version and an online (info) version of the document. The manual is still in a state of flux and will be fleshed out in more detail in later updates.

PLplot 4.0 was primarily the result of efforts by Maurice LeBrun and Geoff Furnish of University of Texas at Austin to extend and improve the previous work (PLplot 2.6b and 3.0, by Tony Richardson). PLplot 4.0 is currently used as the main graphics engine for TPC (Toroidal Particle Code), a large plasma simulation code developed at the IFS [**?**]. During this work we have found that PLplot compares well with "heavier" packages (read: expensive, slow) and is an excellent choice for scientists seeking an inexpensive (or free) but high quality graphics package that runs on many different computing platforms.

Some of the improvements in PLplot 4.0 include: the addition of several new routines to enhance usage from Fortran and design of a portable C to Fortran interface. Additional support was added for coordinate mappings in contour plots and some bugs fixed. New labelling options were added. The font handling code was made more flexible and portable. A portable PLplot metafile driver and renderer was developed, allowing one to create a generic graphics file and do the actual rendering later (even on a different system). The ability to create family output files was added. The internal code structure was dramatically reworked, with elimination of global variables (for a more robust package), the drivers rewritten to improve consistency, and the ability to maintain multiple output streams added. An XFig driver was added. Other contributions include Clair Nielsen's (LANL) X-window driver (very nice for high-speed color graphics) and tektronix file viewer. At present, Maurice LeBrun and Geoff Furnish are the active developers and maintainers of PLplot.

We have attempted to keep PLplot 4.0 backward compatible with previous versions of PLplot. However, some functions are now obsolete, and many new ones have been added (e.g. new contouring functions, variable get/set routines, functions that affect label appearance). Codes written in C that use PLplot must be recompiled including the new header file (`plplot.h`) before linking to the new PLplot library.

PLplot is currently known to work on the following systems: SUNOS, HP-UX, A/IX, DG/UX, UNICOS, Ultrix, VMS, Amiga/Exec, MS-DOS, OS/2, and NeXT, with more expected. The PLplot package is freely distributable, but *not* in the public domain. See section 1.5 for distribution criteria.

We welcome suggestions on how to improve this code, especially in the form of user-contributed enhancements or bug fixes. If PLplot is used in any published papers, please include an acknowledgment or citation of our work, which will help us to continue improving PLplot. Please direct all communication to:

Dr. Maurice LeBrun	mjl@fusion.ph.utexas.edu
Institute for Fusion Studies	University of Texas
Geoff Furnish	furnish@fusion.ph.utexas.edu
Institute for Fusion Studies	University of Texas

The original version of this manual was written by Sze Tan.

1.2 Getting a copy of the PLplot package

At present, the only mechanism we are providing for distribution of the PLplot is by electronic transmission over the Internet. We encourage others to make it available to users without Internet access. PLplot may be obtained by `ftp` from `dino.ph.utexas.edu` (128.83.179.86). Login as user `anonymous`, set file transfer type to binary, and get the newest PLplot archive in the `plplot/` subdirectory. We will provide PLplot in both `zip` and `tar.gz` archives; get whichever one you prefer.

1.3 Installing and Using the PLplot Library

OLD DOCS, NEEDS UPDATING!

The installation procedure is by necessity system specific; installation notes for each system are provided in Appendix **??**. The procedure requires that all of the routines be compiled and they are then usually placed in a linkable library.

After the library has been created, you can write your main program to make the desired PLplot calls. Example programs in both C and Fortran are included as a guide (if calling from C, you must include `plplot.h` into your program; see Appendix **??** for more details). Plots generated from the example programs are shown at the end of this work.

You will then need to compile your program and link it with the PLplot library(s). Again, please refer to the documentation specific to your system for this. Note that there may be more than one library available to link with, such as single or double precision, with or without X window libraries, IEEE floating point or Motorola FFP, etc. Make sure you link to the correct one for your program.

1.4 Organization of this Manual

OLD DOCS, NEEDS UPDATING!

The PLplot library has been designed so that it is easy to write programs producing graphical output without having to set up large numbers of parameters. However, more precise control of the results may be necessary, and these are accomodated by providing lower-level routines which change the system defaults. The manual first describs the overall process of producing a graph using the high-level routines (see 2.1 [Simple Plot], page 11). For a discussion of the underlying concepts of the plotting process and an introduction to some of the more complex routines see 3 [Advanced Use], page 17. An alphabetical list of the user-accessible PLplot functions with detailed descriptions is given in the reference section of the manual (see 6 [API], page 45).

Because the PLplot kernel is written in C, standard C syntax is used in the description of each PLplot function. The C and Fortran language interfaces are discussed in Appendix **??**; look there if you have difficulty interpreting the call syntax as described in this manual. The meaning of function (subroutine) arguments is typically the same regardless of whether you are calling from C or Fortran (but there are some exceptions to this). The arguments for each function are usually specified in terms of PLFLT and PLINT — these are the internal PLplot representations for integer and floating point, and are typically a long and a float (or an INTEGER and a REAL, for Fortran programmers). See Appendix **??** for more detail.

Also, you can use PLplot from C++ just as you would from C. No special classes are available at this time, just use it as any other procedural type library. Simply include `plplot.h`, and invoke as you would from C.

The output devices supported by PLplot are listed in Appendix **??**, along with description of the device driver–PLplot interface, metafile output, family files, and vt100/tek4010 emulators. In Appendix**??** the usage and installation for each system supported by PLplot is described (not guaranteed to be entirely up-to-date; check the release notes to be sure).

1.5 Copyrights

The PLplot package may be distributed under the terms of the GNU General Public License (GPL), except as noted below. The text of this license is given in the file COPYING in the distribution directory.

The PLplot library may be distributed under the terms of the GNU Library General Public License (LGPL), except as noted below. The text of this license is given in the file COPYING.LIB in the distribution directory.

The startup code used in argument handling (utils/plrender.c and src/plargs.c) is partially derived from "xterm.c" of the X11R5 distribution, and its copyright is reproduced here:

in supporting documentation, and that the names of Digital or MIT not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

DIGITAL DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL DIGITAL BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE. **

Any file that is explicitly marked as "public domain" is free from any restriction on distribution.

Any file that has a explicit copyright notice may be distributed under the terms of both the GPL and whatever stated conditions accompany the copyright.

Notes: the intent behind these licenses is to ensure that PLplot continues to evolve in a positive way, while remaining freely distributable. Placing the library under the LGPL enables software developers to use the PLplot library as an integral part of their product, even a commercial product. Under the conditions of the LGPL, however, the source code to the library must remain freely available. This means that if you modify the library to better suit your purposes, you are required to distribute those changes freely to the general PLplot user community.

1.6 Credits

PLplot 5.0 was created through the effort of many individuals and funding agencies. We would like to acknowledge the support (financial and otherwise) of the following institutions:

Thanks are also due to the many contributors to PLplot, including:

Finally, thanks to all those who submitted bug reports and other suggestions.

Part II

Programming

Chapter 2

Simple Use of PLplot

This is the simple introduction to plotting with plplot.

2.1 Plotting a Simple Graph

We shall first consider plotting simple graphs showing the dependence of one variable upon another. Such a graph may be composed of several elements:

- A box which defines the ranges of the variables, perhaps with axes and numeric labels along its edges.
- A set of points or lines within the box showing the functional dependence.
- A set of labels for the variables and a title for the graph.

In order to draw such a graph, it is necessary to call at least four of the PLplot functions:

1. **`plinit`**, to initialize PLplot.
2. **`plenv`**, to define the range and scale of the graph, and draw labels, axes, etc.
3. One or more calls to **`plline`** or **`plpoin`** to draw lines or points as needed. Other more complex routines include **`plbin`** and **`plhist`** to draw histograms, **`plerrx`** and **`plerry`** to draw error-bars.
4. **`plend`**, to close the plot.

More than one graph can be drawn on a single set of axes by making repeated calls to the routines listed in item 3 above. PLplot only needs to be initialized once unless plotting to multiple output devices.

2.2 Initializing PLplot

Before any actual plotting calls are made, a graphics program must call **plinit** (page 62), is the main initialization routine for PLplot. It sets up all internal data structures necessary for plotting and initializes the output device driver. If the output device has not already been specified when **plinit** is called, a list of valid output devices is given and the user is prompted for a choice. Either the device number or a device keyword is accepted.

There are several routines affecting the initialization that must be called *before* **plinit**, if they are used. The function **plsdev** (page **??**) allows you to set the device explicitly. The function **plsdev** (page **??**) may be called to divide the output device plotting area into several subpages of equal size, each of which can be used separately.

One advances to the next page (or screen) via **pladv** (page 45). If subpages are used, this can be used to advance to the next subpage or to a particular subpage.

2.3 Defining Plot Scales and Axes

The function **plenv** is used to define the scales and axes for simple graphs. **plenv** starts a new picture on the next subpage (or a new page if necessary), and defines the ranges of the variables required. The routine will also draw a box, axes, and numeric labels if requested. The syntax for **plenv** is:

```
plenv(xmin, xmax, ymin, ymax, just, axis)
```

xmin, xmax (PLFLT, input):
: The left and right limits for the horizontal axis.

ymin, ymax (PLFLT, input):
: The bottom and top limits for the vertical axis.

just (PLINT, input):
: This should be zero or one. If **just** is one, the scales of the x-axis and y-axis will be the same (in units per millimeter); otherwise the axes are scaled independently. This parameter is useful for ensuring that objects such as circles have the correct aspect ratio in the final plot.

axis (PLINT, input):
: **axis** controls whether a box, tick marks, labels, axes, and/or a grid are drawn.

- **axis = -2**: No box or annotation.
- **axis = -1**: Draw box only.
- **axis = 0**: Draw box, labelled with coordinate values around edge.
- **axis = 1**: In addition to box and labels, draw the two axes X = 0 and Y = 0.

- `axis = 2`: Same as `axis = 1`, but also draw a grid at the major tick interval.
- `axis = 10`: Logarithmic X axis, linear Y axis.
- `axis = 11`: Logarithmic X axis, linear Y axis and draw line Y = 0.
- `axis = 20`: Linear X axis, logarithmic Y axis.
- `axis = 21`: Linear X axis, logarithmic Y axis and draw line X = 0.
- `axis = 30`: Logarithmic X and Y axes.

Note: Logarithmic axes only affect the appearance of the axes and their labels, so it is up to the user to compute the logarithms prior to passing them to **plenv** and any of the other routines. Thus, if a graph has a 3-cycle logarithmic axis from 1 to 1000, we need to set **xmin** = **log10**(1) = 0.0, and **xmax** = **log10**(1000) = 3.0.

For greater control over the size of the plots, axis labelling and tick intervals, more complex graphs should make use of the functions **plvpor** (page 76), **plvasp** (page 75), **plvpas** (page 75), **plwind** (page 78), **plbox** (page 46), and routines for manipulating axis labelling **plgxax** (page 60) through **plszax** (page 75).

2.4 Labelling the Graph

The function **pllab** may be called after **plenv** to write labels on the x and y axes, and at the top of the picture. All the variables are character variables or constants. Trailing spaces are removed and the label is centered in the appropriate field. The syntax for **pllab** is:

pllab(xlbl, ylbl, toplbl)

xlbl (char *, input):
Pointer to string with label for the X-axis (bottom of graph).

ylbl (char *, input):
Pointer to string with label for the Y-axis (left of graph).

toplbl (char *, input):
Pointer to string with label for the plot (top of picture).

More complex labels can be drawn using the function **plmtex** (page 64). For discussion of writing text in a plot see 2.5.3 [Text], page 15, and for more detailed discussion about label generation see 3.6 [Annotation], page 22.

2.5 Drawing the Graph

PLplot can draw graphs consisting of points with optional error bars, line segments or histograms. Functions which perform each of these actions may be called after setting up the plotting environment using **plenv**. All of the following functions draw within the box defined by **plenv**, and any lines crossing the boundary are clipped. Functions are also provided for drawing surface and contour representations of multi-dimensional functions. See 3 [Advanced Use], page 17 for discussion of finer control of plot generation.

2.5.1 Drawing Points

plpoin and **plsym** mark out `n` points `(x[i], y[i])` with the specified symbol. The routines differ only in the interpretation of the symbol codes. **plpoin** uses an extended ASCII representation, with the printable ASCII codes mapping to the respective characters in the current font, and the codes from 0–31 mapping to various useful symbols. In **plsym** however, the code is a Hershey font code number. Example programs are provided which display each of the symbols available using these routines.

`plpoin(n, x, y, code)`
`plsym(n, x, y, code)`

`n` (PLINT, input):
: The number of points to plot.

`x, y` (PLFLT *, input):
: Pointers to arrays of the coordinates of the `n` points.

`code` (PLINT, input):
: Code number of symbol to draw.

2.5.2 Drawing Lines or Curves

PLplot provides two functions for drawing line graphs. All lines are drawn in the currently selected color, style and width. See 3.8 [Line Attributes], page 24 for information about changing these parameters.

plline draws a line or curve. The curve consists of `n-1` line segments joining the `n` points in the input arrays. For single line segments, **pljoin** is used to join two points.

`plline(n, x, y)`

`n` (PLINT, input):
: The number of points.

`x, y` (PLFLT *, input):
: Pointers to arrays with coordinates of the `n` points.

`pljoin(x1, y1, x2, y2)`

`x1, y1` (PLFLT, input):
: Coordinates of the first point.

`x2, y2` (PLFLT, input):
: Coordinates of the second point.

2.5.3 Writing Text on a Graph

`plptex` allows text to be written within the limits set by `plenv`. The reference point of a text string may be located anywhere along an imaginary horizontal line passing through the string at half the height of a capital letter. The parameter `just` specifies where along this line the reference point is located. The string is then rotated about the reference point through an angle specified by the parameters `dx` and `dy`, so that the string becomes parallel to a line joining `(x, y)` to `(x+dx, y+dy)`.

```
plptex(x, y, dx, dy, just, text)
```

`x, y` (PLFLT, input):
: Coordinates of the reference point.

`dx, dy` (PLFLT, input):
: These specify the angle at which the text is to be printed. The text is written parallel to a line joining the points `(x, y)` to `(x+dx, y+dy)` on the graph.

`just` (PLFLT, input):
: Determines justification of the string by specifying which point within the string is placed at the reference point `(x, y)`. This parameter is a fraction of the distance along the string. Thus if `just = 0.0`, the reference point is at the left-hand edge of the string. If `just = 0.5`, it is at the center and if `just = 1.0`, it is at the right-hand edge.

`text` (char *, input):
: Pointer to the string of characters to be written.

2.5.4 Area Fills

Area fills are done in the currently selected color, line style, line width and pattern style.

`plfill` fills a polygon. The polygon consists of `n` vertices which define the polygon.

```
plfill(n, x, y)
```

`n` (PLINT, input):
: The number of vertices.

`x, y` (PLFLT *, input):
: Pointers to arrays with coordinates of the `n` vertices.

2.5.5 More Complex Graphs

Functions **plbin** (page 45) and **plhist** (page 61) are provided for drawing histograms, and functions **plerrx** (page 56) and **plerry** (page 56) draw error bars about specified points. There are lots more too (see 6 [API], page 45).

2.6 Finishing Up

Before the end of the program, *always* call **plend** (page 55) to close any output plot files and to free up resources. For devices that have separate graphics and text modes, **plend** resets the device to text mode.

2.7 In Case of Error

If a fatal error is encountered during execution of a PLplot routine then **plexit** (page 57) is called. This routine prints an error message, does resource recovery, and then exits. The user may specify an error handler via **plsexit** (page **??**) that gets called before anything else is done, allowing either the user to abort the error termination, or clean up user-specific data structures before exit.

Chapter 3

Advanced Use of PLplot

In this chapter, we describe more precisely how to control the position and scaling of a graph, how to alter the low-level line and character attributes, and how to use the functions in PLplot for drawing three-dimensional surface plots and contour plots.

Still updating this chapter.

3.1 Basic PLplot Concepts

When drawing a graph, the programmer usually wishes to specify the coordinates of the points to be plotted in terms of the values of the variables involved. These coordinates are called *world coordinates*, and may have any floating-point value representable by the computer. The *window* refers to the rectangular region of world-coordinate space which is to be graphed. This window is mapped onto a rectangular region of the *view surface*, which can be a terminal screen or a sheet of paper in the output device (or a portion thereof, if subpages are being used). This physical region onto which the window is mapped is called the *viewport*. Before a graph can be drawn, the program must define both the window and the viewport by calling appropriate routines in PLplot.

3.2 Specifying the Output Device

The main initialization routine for PLplot is `plinit`, which sets up all internal data structures necessary for plotting and initializes the output device driver. The output device can be a terminal, disk file, window system, pipe, or socket. If the output device has not already been specified when `plinit` is called, a list of valid output devices is given and the user is prompted for a choice. For example:

```
% x01c

Plotting Options:
 < 1> xwin       X-Window (Xlib)
 < 2> dp         Tcl-DP/TK Window
 < 3> tk         Tcl/TK Window
```

```
 < 4> xterm      Xterm Window
 < 5> tekt       Tektronix Terminal (4010)
 < 6> tek4107t   Tektronix Terminal (4105/4107)
 < 7> mskermit   MS-Kermit emulator
 < 8> versaterm  Versaterm vt100/tek emulator
 < 9> vlt        VLT vt100/tek emulator
 <10> plmeta     PLPLOT Native Meta-File
 <11> tekf       Tektronix File (4010)
 <12> tek4107f   Tektronix File (4105/4107)
 <13> ps         PostScript File (monochrome)
 <14> psc        PostScript File (color)
 <15> xfig       Xfig file
 <16> ljiip      LaserJet IIp/deskjet compressed graphics
 <17> ljii       LaserJet II Bitmap File (150 dpi)
 <18> null       Null device

Enter device number or keyword:
```

Either the device number or a device keyword is accepted. Specifying the device by keyword is preferable in aliases or scripts since the device number is dependent on the install procedure (the installer can choose which device drivers to include). The device can be specified prior to the call to **plinit** by:

- A call to **plsdev** (page **??**).
- The **-dev** *device* command line argument, if the program's command line arguments are being passed to the PLplot function **plParseInternalOpts** (page **??**).

Additional startup routines **plstar** (page 72) and **plstart** (page 72) are available but these are simply front-ends to **plinit**, and should be avoided. It is preferable to call **plinit** directly, along with the appropriate setup calls, for the greater amount of control this provides (see the example programs for more info).

Before **plinit** is called, you may modify the number of subpages the output device is divided into via a call to **plssub** (page **??**). Subpages are useful for placing several graphs on a page, but all subpages are constrained to be of the same size. For greater flexibility, viewports can be used (see 3.4 [Viewports], page 20 for more info on viewports). The routine **pladv** is used to advance to a particular subpage or to the next subpage. The screen is cleared (or a new piece of paper loaded) if a new subpage is requested when there are no subpages left on the current page. When a page is divided into subpages, the default character, symbol and tick sizes are scaled inversely as the square root of the number of subpages in the vertical direction. This is designed to improve readability of plot labels as the plot size shrinks.

PLplot has the ability to write to multiple output streams. An output stream corresponds to a single logical device to which one plots independent of all other streams. The function **plsstrm** (page 71) is used to switch between streams – you may only write to one output stream at a time. At present, an output stream is not limited by the type of device, however, it may not be wise to attempt opening two terminal devices. An example usage for the creation of multiple streams is as follows:

```
#include "plplot.h"

main()
{
    int nx = 2, ny = 2;

    plssub(nx, ny);
    plsdev("xwin");
    plinit();

    <plots for stream 0>

    plsstrm(1);
    plssub(nx, ny);
    plsdev("plmeta");
    plsfnam("tst.plm");
    plinit();

    <plots for stream 1>

    plsstrm(0);

    <plots for stream 0>
```

and so on, for sending output simultaneously to an X-window and a metafile. The default stream corresponds to stream number zero. At present, the majority of output drivers can only be used by a single stream (exceptions include the metafile driver and X-window driver). Also see example program 14 (note: only the C version is available, although it can be done equally well from Fortran).

At the end of a plotting program, it is important to close the plotting device by calling **plend** (page 55). This flushes any internal buffers and frees any memory that may have been allocated, for all open output streams. You may call **plend1** (page 55) to close the plotting device for the current output stream only. Note that if PLplot is initialized more than once during a program to change the output device, an automatic call to **plend1** is made before the new device is opened for the given stream.

3.3 Command line arguments

PLplot supports a large number of command line arguments, but it is up to the user to pass these to PLplot for processing at the beginning of execution. The function **plParseInternalOpts** (page **??**) is responsible for parsing the argument list, removing all that are recognized by

PLplot, and taking the appropriate action before returning. There are an extensive number of options available to affect this process. The command line arguments recognized by PLplot are given by the -h option:

```
% x01c -h

Usage:
        x01c [plplot options]

plplot options:
    -h                  Print out this message
    -v                  Print out the plplot library version number
    -dev name           Output device name
    -o name             Output filename
    -display name       X server to contact
    -px number          Plots per page in x
    -py number          Plots per page in y
    -geometry geom      Window size, in pixels (e.g. -geometry 400x300)
    -wplt xl,yl,xr,yr   Relative coordinates [0-1] of window into plot
    -mar margin         Margin space in relative coordinates (0 to 0.5, def 0)
    -a aspect           Page aspect ratio (def: same as output device)
    -jx justx           Page justification in x (-0.5 to 0.5, def 0)
    -jy justy           Page justification in y (-0.5 to 0.5, def 0)
    -ori orient         Plot orientation (0,2=landscape, 1,3=portrait)
    -width width        Sets pen width (1 <= width <= 10)
    -bg color           Background color (0=black, FFFFFF=white)
    -fam                Create a family of output files
    -fsiz size          Output family file size in MB (e.g. -fsiz 1.0)
    -fbeg number        First family member number on output
    -finc number        Increment between family members
    -fflen length       Family member number minimum field width
    -nopixmap           Don't use pixmaps in X-based drivers
    -db                 Double buffer X window output
    -np                 No pause between pages
    -server_name name   Main window name of plplot server (tk driver)
    -server_host name   Host to run plplot server on (dp driver)
    -server_port name   Port to talk to plplot server on (dp driver)
    -user name          User name on remote node (dp driver)

All parameters must be white-space delimited.  Some options are driver
dependent.  Please see the plplot reference document for more detail.
```

3.4 Defining the Viewport

After defining the view surface, it is necessary to define the portion of this surface which is to be used for plotting the graph (the viewport). All lines and symbols (except for labels drawn by **plbox**, **plmtex** and **pllab**) are clipped at the viewport boundaries.

Viewports are created within the current subpage. If the division of the output device into equally sized subpages is inappropriate, it is best to specify only a single subpage which occupies the entire output device (by setting **nx = 1** and **ny = 1** in **plstart** or **plstar**), and use one of the viewport specification subroutines below to place the plot in the desired position on the page.

There are four methods for specifying the viewport size, using the subroutines **plvpor** (page 76), **plsvpa** (page 73), **plvasp** (page 75), and **plvpas** (page 75). Each of these has the format:

```
plvpor(xmin, xmax, ymin, ymax);
plsvpa(xmin, xmax, ymin, ymax);
plvasp(aspect);
plvpas(xmin, xmax, ymin, ymax, aspect);
```

where in the case of **plvpor** and **plvpas**, the arguments are given in *normalized subpage coordinates* which are defined to run from 0.0 to 1.0 along each edge of the subpage. Thus for example,

```
plvpor(0.0, 0.5, 0.5, 1.0);
```

uses the top left quarter of the current subpage.

In order to get a graph of known physical size, the routine **plsvpa** defines the viewport in terms of absolute coordinates (millimeters) measured from the bottom left-hand corner of the current subpage. This routine should only be used when the size of the view surface is known, and a definite scaling is required.

The routine **plvasp** gives the largest viewport with the given aspect ratio that fits in the current subpage (i.e. the ratio of the length of the y axis to that of the x axis is equal to **aspect**). It also allocates space on the left and top of the viewport for labels.

The routine **plvpas** gives the largest viewport with the given aspect ratio that fits in the specified region (specified with normalized subpage coordinates, as with **plvpor**). This routine is functionally equivalent to **plvpor** when a “natural” aspect ratio is chosen (done by setting **aspect** to 0.0). Unlike **plvasp**, this routine reserves no extra space at the edges for labels.

To help the user call **plsvpa** correctly, the routine **plgspa** (page 60) is provided which returns the positions of the extremities of the current subpage measured in millimeters from the bottom left-hand corner of the device. Thus, if to set up a viewport with a 10.0 mm margin around it within the current subpage, the following sequence of calls may be used:

```
plgspa(xmin, xmax, ymin, ymax);
plsvpa(10.0, xmax-xmin-10.0, 10.0, ymax-ymin-10.0);
```

A further routine **plvsta** (page 76) is available which sets up a standard viewport within the current subpage with suitable margins on each side of the viewport. This may be used for simple graphs, as it leaves enough room for axis labels and a title. This standard viewport is that used by **plenv** (see 3.7 [Standard Window], page 23).

Another way to get a specified aspect ratio is via the routine **plsasp** [not!.. fix this], which sets the global aspect ratio and must be called prior to **plstar**. An aspect ratio of 0.0 corresponds to "natural" dimensions (i.e. fill the page); any positive value will give the specified aspect ratio. This scaling of plots is actually done in the driver, and so may not work for all output devices (note that **plrender** is capable of scaled aspect ratio plots to any device whether that device supports scaling or not). In such scaled plots, absolute plotting is done in the scaled coordinate system.

3.5 Defining the Window

The window must be defined after the viewport in order to map the world coordinate rectangle into the viewport rectangle. The routine **plwind** (page 78) is used to specify the rectangle in world-coordinate space. For example, if we wish to plot a graph showing the collector current

I_C as a function of the collector to emitter voltage V_{CE} for a transistor where $0 \le I_C \le 10.0$ mA and $0 \le V_{CE} \le 12.0$ V, we would call the function **plwind** as follows:

```
plwind(0.0, 12.0, 0.0, 10.0);
```

Note that each of the arguments is a floating point number, and so the decimal points are required. If the order of either the X limits or Y limits is reversed, the corresponding axis will point in the opposite sense, (i.e., right to left for X and top to bottom for Y). The window must be defined before any calls to the routines which actually draw the data points. Note however that **plwind** may also be called to change the window at any time. This will affect the appearance of objects drawn later in the program, and is useful for drawing two or more graphs with different axes on the same piece of paper.

3.6 Annotating the Viewport

The routine **plbox** (page 46) is used to specify whether a frame is drawn around the viewport and to control the positions of the axis subdivisions and numeric labels. For our simple graph of the transistor characteristics, we may wish to draw a frame consisting of lines on all four sides of the viewport, and to place numeric labels along the bottom and left hand side. We can also tell PLplot to choose a suitable tick interval and the number of subticks between the major divisions based upon the data range specified to **plwind**. This is done using the following statement

```
plbox("bcnst", 0.0, 0, "bcnstv", 0.0, 0);
```

Another routine **pllab** (page 62) provides for text labels for the bottom, left hand side and top of the viewport. These labels are not clipped, even though they lie outside the viewport (but they are clipped at the subpage boundaries). **pllab** actually calls the more general routine **plmtex** which can be used for plotting labels at any point relative to the viewport. For our example, we may use

```
pllab("V#dCE#u (Volts)", "I#dC#u (mA)", "TRANSISTOR CHARACTERISTICS");
```

Note that **#d** and **#u** are escape sequences (see 3.11 [Escape Sequences], page 26) which allow subscripts and superscripts to be used in text. They are described more fully later in this chapter.

The appearance of axis labels may be further altered by auxiliary calls to **plprec** (page 67), **plschr** (page 68), **plsxax** (page 74), **plsyax** (page 74), and **plszax** (page 75). The routine **plprec** is used to set the number of decimal places precision for axis labels, while **plschr** modifies the heights of characters used for the axis and graph labels. Routines **plsxax**, **plsyax**, and **plszax** are used to modify the **digmax** setting for each axis, which affects how floating point labels are formatted.

The **digmax** variable represents the maximum field width for the numeric labels on an axis (ignored if less than one). If the numeric labels as generated by PLplot exceed this width, then PLplot automatically switches to floating point representation. In this case the exponent will be placed at the top left for a vertical axis on the left, top right for a vertical axis on the right, and bottom right for a horizontal axis.

The determination of maximum length (i.e. **digmax**) for fixed point quantities is complicated by the fact that long fixed point representations look much worse than the same sized floating point representation. Further, a fixed point number with magnitude much less than one will actually gain in precision when written as floating point. There is some compensation for this effect built into PLplot, thus the internal representation for number of digits kept (**digfix**) may not always match the user's specfication (via **digmax**). However,

Finally, after the call to **plbox**, the user may call routines **plgxax** (page 60), **plgyax** (page 61), or **plgzax** (page 61) to obtain information about the window just drawn. This can be helpful when deciding where to put captions. For example, a typical usage would be to call **plgyax** to get the value of **digits**, then offset the y axis caption by that amount (plus a bit more) so that the caption "floats" just to the outside of the numeric labels. Note that the **digits** value for each axis for the current plot is not correct until *after* the call to **plbox** is complete.

3.7 Setting up a Standard Window

Having to call **pladv**, **plvpor**, **plwind** and **plbox** is excessively cumbersome for drawing simple graphs. Subroutine **plenv** combines all four of these in one subroutine, using the standard viewport, and a limited subset of the capabilities of **plbox**. For example, the graph described above could be initiated by the call:

```
plenv(0.0, 12.0, 0.0, 10.0, 0, 0);
```

which is equivalent to the following series of calls:

```
pladv(0);
plvsta();
plwind(0.0, 12.0, 0.0, 10.0);
plbox("bcnst", 0.0, 0, "bcnstv", 0.0, 0);
```

3.8 Setting Line Attributes

The graph drawing routines may be freely mixed with those described in this section, allowing the user to control line color, width and styles. The attributes set up by these routines apply modally, i.e, all subsequent objects (lines, characters and symbols) plotted until the next change in attributes are affected in the same way. The only exception to this rule is that characters and symbols are not affected by a change in the line style, but are always drawn using a continuous line.

Line color is set using the routine `plcol` (page 49). The argument is ignored for devices which can only plot in one color, although some terminals support line erasure by plotting in color zero.

Line width is set using `plwid` (page 77). This option is not supported by all devices.

Line style is set using the routine `plstyl` (page 73) or `pllsty` (page 63). A broken line is specified in terms of a repeated pattern consisting of marks (pen down) and spaces (pen up). The arguments to this routine are the number of elements in the line, followed by two pointers to integer arrays specifying the mark and space lengths in micrometers. Thus a line consisting of long and short dashes of lengths 4 mm and 2 mm, separated by spaces of length 1.5 mm is specified by:

```
mark[0] = 4000;
mark[1] = 2000;
space[0] = 1500;
space[1] = 1500;
plstyl(2, mark, space);
```

To return to a continuous line, just call `plstyl` with first argument set to zero. You can use `pllsty` to choose between 8 different predefined styles.

3.9 Setting the Area Fill Pattern

The routine `plpat` (page 66) can be used to set the area fill pattern. The pattern consists of 1 or 2 sets of parallel lines with specified inclinations and spacings. The arguments to this routine are the number of sets to use (1 or 2) followed by two pointers to integer arrays (of 1 or 2 elements) specifying the inclinations in tenths of a degree and the spacing in micrometers (the inclination should be between -900 and 900). Thus to specify an area fill pattern consisting of horizontal lines spaced 2 mm apart use:

```
*inc = 0;
*del = 2000;
plpat(1, inc, del);
```

To set up a symmetrical crosshatch pattern with lines directed 30 degrees above and below the horizontal and spaced 1.5 mm apart use:

```
*inc = 300;
*(inc+1) = -300;
*del = 1500;
*(del+1) = 1500;
plpat(2, inc, del);
```

The routine **plpsty** (page 67) can be used to select from 1 of 8 predefined patterns.

The area fill routines also use the current line style, width and colors to give a virtually infinite number of different patterns.

3.10 Setting Character and Symbol Attributes

There are two character sets included with PLplot. These are known as the standard and extended character sets respectively. The standard character set is a subset of the extended set. It contains 177 characters including the ascii characters in a normal style font, the greek alphabet and several plotter symbols. The extended character set contains almost 1000 characters, including four font styles, and several math, musical and plotter symbols.

The standard character set is loaded into memory automatically when **plstar** or **plstart** is called. The extended character set is loaded by calling **plfontld** (page 58). The extended character set requires about 50 kbytes of memory, versus about 5 kbytes for the standard set. **plfontld** can be used to switch between the extended and standard sets (one set is unloaded before the next is loaded). **plfontld** can be called before **plstar**.

When the extended character set is loaded there are four different font styles to choose from. In this case, the routine **plfont** (page 57) sets up the default font for all character strings. It may be overridden for any portion of a string by using an escape sequence within the text, as described below. This routine has no effect when the standard font set is loaded. The default font (1) is simple and fastest to draw; the others are useful for presentation plots on a high-resolution device.

The font codes are interpreted as follows:

- `font = 1`: normal simple font
- `font = 2`: roman font
- `font = 3`: italic font
- `font = 4`: script font

The routine **plschr** (page 68) is used to set up the size of subsequent characters drawn. The actual height of a character is the product of the default character size and a scaling factor. If no call is made to **plschr**, the default character size is set up depending on the number of subpages defined in the call to **plstar** or **plstart**, and the scale is set to 1.0. Under normal circumstances, it is recommended that the user does not alter the default height, but simply use the scale parameter. This can be done by calling **plschr** with **def = 0.0** and **scale** set to the desired multiple of the default height. If the default height is to be changed, **def** is set to the new default height in millimeters, and the new character height is again set to **def** multiplied by **scale**.

The routine **plssym** (page 71) sets up the size of all subsequent symbols drawn by calls to **plpoin** and **plsym**. It operates analogously to **plschr** as described above.

The lengths of major and minor ticks on the axes are set up by the routines **plsmaj** (page 70) and **plsmin** (page 70).

3.11 Escape Sequences in Text

The routines which draw text all allow you to include escape sequences in the text to be plotted. These are character sequences that are interpreted as instructions to change fonts, draw superscripts and subscripts, draw non-ASCII (e.g. Greek), and so on. All escape sequences start with a number symbol (**#**).

The following escape sequences are defined:

- **#u**: move up to the superscript position (ended with **#d**)
- **#d**: move down to subscript position (ended with **#u**)
- **#b**: backspace (to allow overprinting)
- **##**: number symbol
- **#+**: toggle overline mode
- **#-**: toggle underline mode
- **#gx**: Greek letter corresponding to Roman letter **x** (see below)
- **#fn**: switch to normal font
- **#fr**: switch to Roman font
- **#fi**: switch to italic font
- **#fs**: switch to script font
- **#(nnn)**: Hershey character nnn (1 to 4 decimal digits)

Sections of text can have an underline or overline appended. For example, the string

$$\overline{\mathrm{S}}(\underline{\mathrm{freq}})$$

is obtained by specifying "#+S#+(#-freq#-)".

Greek letters are obtained by #g followed by a Roman letter. Table ?? shows how these letters map into Greek characters.table

3.12 Three Dimensional Surface Plots

PLplot includes routines that will represent a single-valued function of two variables as a surface. In this section, we shall assume that the function to be plotted is Z[X][Y], where Z represents the dependent variable and X and Y represent the independent variables.

As usual, we would like to refer to a three dimensional point (X, Y, Z) in terms of some meaningful user-specified coordinate system. These are called *three-dimensional world coordinates*. We need to specify the ranges of these coordinates, so that the entire surface is contained within the cuboid defined by xmin<x<xmax, ymin<y<ymax and zmin<z<zmax. Typically, we shall want to view the surface from a variety of angles, and to facilitate this, a two-stage mapping of the enclosing cuboid is performed. Firstly, it is mapped into another cuboid called the *normalized box* whose size must also be specified by the user, and secondly this normalized box is viewed from a particular azimuth and elevation so that it can be projected onto the two-dimensional window.

This two-stage transformation process allows considerable flexibility in specifying how the surface is depicted. The lengths of the sides of the normalized box are independent of the world coordinate ranges of each of the variables, making it possible to use "reasonable" viewing angles even if the ranges of the world coordinates on the axes are very different. The size of the normalized box is determined essentially by the size of the two-dimensional window into which it is to be mapped. The normalized box is centered about the origin in the x and y directions, but rests on the plane z = 0. It is viewed by an observer located at altitude alt and azimuth az, where both angles are measured in degrees. The altitude should be restricted to the range zero to ninety degrees for proper operation, and represents the viewing angle above the xy plane. The azimuth is defined so that when az = 0, the observer sees the xz plane face on, and as the angle is increased, the observer moves clockwise around the box as viewed from above the xy plane. The azimuth can take on any value.

The first step in drawing a surface plot is to decide on the size of the two-dimensional window and the normalized box. For example, we could choose the normalized box to have sides of length

```
basex = 2.0;
basey = 4.0;
height = 3.0;
```

A reasonable range for the x coordinate of the two-dimensional window is -2.5 to +2.5, since the length of the diagonal across the base of the normalized box is $\sqrt{2^2+4^2} = 2\sqrt{5}$, which fits into this coordinate range. A reasonable range for the y coordinate of the two dimensional window in this case is -2.5 to +4, as the the projection of the normalized box lies in this range for the allowed range of viewing angles.

The routine **plwind** (page 78) or **plenv** (page 55) is used in the usual way to establish the size of the two-dimensional window. The routine **plw3d** (page 76) must then be called to establish the range of the three dimensional world coordinates, the size of the normalized box and the viewing angles. After calling **plw3d**, the actual surface is drawn by a call to **plot3d** (page 65).

For example, if the three-dimensional world-coordinate ranges are
$-10.0 \leq \mathbf{x} \leq 10.0$, $-3.0 \leq \mathbf{y} \leq +7.0$ and $0.0 \leq \mathbf{z} \leq 8.0$, we could use the following statements:

```
xmin2d = -2.5;
xmax2d =  2.5;
ymin2d = -2.5;
ymax2d =  4.0;
plenv(xmin2d, xmax2d, ymin2d, ymax2d, 0, -2);
basex = 2.0;
basey = 4.0;
height = 3.0;
xmin = -10.0;
xmax = 10.0;
ymin = -3.0;
ymax = 7.0;
zmin = 0.0;
zmax = 8.0;
alt = 45.0;
az = 30.0;
side = 1;
plw3d(basex, basey, height, xmin, xmax, ymin, ymax, zmin, zmax, alt, az);
plot3d(x, y, z, nx, ny, opt, side);
```

The values of the function are stored in a two-dimensional array `z[][]` where the array element `z[i][j]` contains the value of the function at the point x_i, y_j. (The two-dimensional array `z` is a vectored array instead of a fixed size array. `z` points to an array of pointers which each point to a rwo of the matrix.) Note that the values of the independent variables x_i and y_j do not need to be equally spaced, but they must lie on a rectangular grid. Thus two further arrays `x[nx]` and `y[ny]` are required as arguments to `plot3d` to specify the values of the independent variables. The values in the arrays x and y must be strictly increasing with the index. The argument `opt` specifies how the surface is outlined. If `opt = 1`, a line is drawn representing z as a function of x for each value of y, if `opt = 2`, a line is drawn representing z as a function of y for each value of x, and if `opt = 3`, a net of lines is drawn. The first two options may be preferable if one of the independent variables is to be regarded as a parameter, whilst the third is better for getting an overall picture of the surface. If side is equal to one then sides are drawn on the figure so that the graph doesn't appear to float.

The routine **plmesh** (page 63) is similar to **plot3d**, except that it is used for drawing mesh plots. Mesh plots allow you to see both the top and bottom sides of a surface mesh, while 3D plots allow you to see the top side only (like looking at a solid object). The side option is not available with **plmesh**.

Labelling a three-dimensional or mesh plot is somewhat more complicated than a two dimensional plot due to the need for skewing the characters in the label so that they are parallel to the coordinate axes. The routine **plbox3** (page 47) thus combines the functions of box drawing and labelling.

3.13 Contour and Shade Plots

Several routines are available in PLplot which perform a contour or shade plot of data stored in a two-dimensional array. The contourer uses a contour following algorithm is used, so that it is possible to use non-continuous line styles. Further, one may specify arbitrary coordinate mappings from array indices to world coordinates, such as for contours in a polar coordinate system. In this case it is best to draw the distinction between a C and Fortran language caller, so these are handled in turn.

3.13.1 Contour Plots from C

There are two routines callable from C for plotting contours, **plcont** (page 53) and **plconf** (page 52). Routine **plcont** has the form:

```
plcont(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, pltr)
```

where **z** is the two-dimensional array of size **nx** by **ny** containing samples of the function to be contoured. (**z** is a vectored two-dimensional array as described in the previous section. It is *not* a fixed-size two-dimensional array.) The parameters **kx**, **lx**, **ky** and **ly** specify the portion of **z** that is to be considered. The array **clevel** of length **nlevel** is a list of the desired contour levels. Routine **plconf** has the form:

```
plconf(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, pltr, xg, yg)
```

with all arguments identical to those in **plcont** except for the pointers to transformation arrays **xg** and **yg**. These are used as input to the transformation function **pltr**.

The path of each contour is initially computed in terms of the values of the array indicies which range from 0 to nx-1 in the first index and from 0 to ny-1 in the second index. Before these can be drawn in the current window (see 3.5 [Windows], page 22), it is necessary to convert from these array indicies into world coordinates. This is done by passing a pointer to a user-defined function to **plcont** (the case for **plconf** is slightly different and discussed below). This function pointer is the last argument **pltr**. This function must be declared as type **void** in the module which calls **plcont**. This transformation function must have the parameter list:

```
void pltr(x, y, tx, ty);
```

where (x, y) is the point through which the contour runs expressed in terms of array indicies, and (tx, ty) are pointers to float variables which are the world coordinates of the point which corresponds to these indicies.

Often, the transformation between array indices and world coordinates can be expressed as a linear transformation. A routine is provided within the library which can be passed to **plcont** as **pltr**. This transformation routine is:

```
#include "plplot.h"

#ifdef PLSTDC
void
xform(PLFLT x, PLFLT y, PLFLT * tx, PLFLT * ty)
#else
void
xform(x, y, tx, ty)
PLFLT x, y, *tx, *ty;
#endif
{
    *tx = tr[0] * x + tr[1] * y + tr[2];
    *ty = tr[3] * x + tr[4] * y + tr[5];
}
```

and should be compiled as part of the user's code (it is *not* normally linked in with the PLplot library since the tr data array is unspecified).

Thus by setting up the values in the array tr[], we can apply an arbitrary translation, rotation and/or shear to the array before drawing out the contours. By defining other transformation subroutines, it is possible to draw contours wrapped around polar grids etc. If no transformation is required, you can simply set tr[0] = tr[4] = 1., with other entries set to 0.. Alternately you can pass the built-in function pltr0 to plcont; this function merely sets *tx to x and *ty to y, resulting in no net transformation.

As an example in setting up tr[], suppose that the array z is of size 21 by 41 and contains the values of the function z[x][y], where x ranges from 0.0 to 4.0 and y ranges from -8.0 to 8.0. Furthermore, let us also suppose that the window (as defined using **plenv** or **plwind**) covers this range of world coordinates. Since we wish the index (0, 0) in array z to map to (0.0, -8.0) and the index (20, 40) to map to (4.0, 8.0), and for there to be no skew in the axes, we should choose elements of tr[] so that

$$\begin{aligned} tx &= 0.2x \\ ty &= 0.4y - 8. \end{aligned}$$

and so tr[0] = 0.2, tr[1] = 0.0, tr[2] = 0.0, tr[3] = 0.0, tr[4] = 0.4, tr[5] = -8..

For routine **plconf**, the user must pass pointers to transformation arrays as well as the function pointer. The use of transformation arrays is generally the most convenient and flexible way to map coordinates, since the mapping function (which only handles interpolation) need not change from one problem to the next. In this case there are two built-in transformation functions to assist the user; these are:

```
void pltr1(x, y, tx, ty, xg, yg, nx, ny);
```

```
void pltr2(x, y, tx, ty, xg, yg, nx, ny);
```

where `(xg, yg)` are pointers to the transformation arrays and `(nx, ny)` are the number of gridpoints in each dimension. These functions handle all necessary interpolations to calculate coordinate values between grid points. The transformation arrays passed to `pltr1` must be functions of a single variable only (`xg[i], yg[i]`), while those passed to `pltr2` must be functions of two variables (`xg[i, j], yg[i, j]`). This is further discussed in the following section.

3.13.2 Contour Plots from Fortran

The routines mentioned above are not recommended for use directly from Fortran due to the need to pass a function pointer. That is, the transformation function is written in C and can not generally be changed by the user. The call for routine `plcont` from Fortran is then:

```
call plcont(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, tr)
```

where `tr` is a 6 element array with the coefficients for the transformation function `xform` given above and all other arguments are the same as in the call from C (see above discussion of `plcont`).

The three routines recommended for use from Fortran are `plcon0` (page 50), `plcon1` (page 50), and `plcon2` (page 51). These routines are similar to existing commercial plot package contour plotters in that they offer successively higher complexity, with `plcon0` utilizing no transformation arrays, while those used by `plcon1` and `plcon2` are one and two dimensional, respectively. The call syntax for each is

```
call plcon0(z, nx, ny, kx, lx, ky, ly, clevel, nlevel)
```

```
call plcon1(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, xg1, yg1)
```

```
call plcon2(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, xg2, yg2)
```

The `plcon0` routine is implemented via a call to `plcont` with a very simple (identity) transformation function, while `plcon1` and `plcon2` use interpolating transformation functions in a call to `plconf`.

The transformation arrays are used by these routines to specify a mapping between the computational coordinate system and the physical one. For example, the transformation to polar coordinates might look like:

```
do i = 1, NX
    do j = 1, NY
        xg(i, j) = r(i) * cos( theta(j) )
        yg(i, j) = r(i) * sin( theta(j) )
    enddo
enddo
```

example program 9).

The transformation function not only specifies the transformation at grid points, but also at intermediate locations, via linear interpolation. For example, in the `pltr1` transformation function used by `plcon1`, the 1-d interpolation to get `tx` as a function of `x` looks like (in C):

```
ul = (PLINT)x;
ur = ul + 1;
du = x - ul;

xl = *(xg+ul);
xr = *(xg+ur);

*tx = xl * (1-du)  +  xr * du;
```

while in Fortran this might look like:

```
lxl = x
lxr = lxl + 1
dx = x - lxl

xl = xg(lxl)
xr = xg(lxr)

tx = xl * (1-dx)  +  xr * dx
```

Chapter 4

PLplot Output Drivers

Beginning with PLplot 5.0, the following drivers are supported.

4.1 The Xwin Driver

is cool.

4.2 The Tk Driver

is the prototype of a whole new interaction paradigm. See next chapter.

4.3 The Tcl-DP Driver

is another example of how awesome Tcl is. Useful for many users who wish to use the PLplot programming API, but who do not need to embed PLplot output in a larger Tk interface of their own creation.

This driver ...

4.4 The Postscript Driver

Chapter 5

Building an Extended WISH

Beginning with PLplot 5.0, a new and powerful paradigm for interaction with PLplot is introduced. This new paradigm consists of an integration of PLplot with a powerful scripting language (Tcl), and extensions to that language to support X Windows interface development (Tk) and object oriented programming ([incr Tcl]). Taken together, these four software systems (Tcl/Tk/itcl/PLplot) comprise a powerful environment for the rapid prototyping and development of sophisticated, flexible, X Windows applications with access to the PLplot API. Yet that is only the beginning–Tcl was born to be extended. The true power of this paradigm is achieved when you add your own, powerful, application specific extensions to the above quartet, thus creating an environment for the development of wholly new applications with only a few keystrokes of shell programming ...

5.1 Introduction to Tcl

The Tool Command Language, or just Tcl (pronounced "tickle") is an embedable script language which can be used to control a wide variety of applications. Designed by John Ousterhout of UC Berkeley, Tcl is freely available under the standard Berkeley copyright. Tcl and Tk (described below) are extensively documented in a new book published by Addison Wesley, entitled "Tcl and the Tk toolkit" by John Ousterhout. This book is a must have for those interested in developing powerful extensible applications with high quality X Windows user interfaces. The discussion in this chapter cannot hope to approach the level of introduction provided by that book. Rather we will concentrate on trying to convey some of the excitement, and show the nuts and bolts of using Tcl and some extensions to provide a powerful and flexible interface to the PLplot library within your application.

5.1.1 Motivation for Tcl

The central observation which led Ousterhout to create Tcl was the realization that many applications require the use of some sort of a special purpose, application specific, embedded "macro language". Application programmers cobble these "tiny languages" into their codes in order to provide flexibility and some modicum of high level control. But the end result is frequently a quirky and fragile language. And each application has a different "tiny language" associated with it. The idea behind Tcl, then, was to create a single "core language" which could be easily embedded into a wide variety of applications. Further, it should be easily extensible so that individual applications can easily provide application specific capabilities available in the macro language itself, while still providing a robust, uniform syntax across a variety of applications. To say that Tcl satisfies these requirements would be a spectacular understatement.

5.1.2 Capabilities of Tcl

The mechanics of using Tcl are very straightforward. Basically you just have to include the file '`tcl.h`', issue some API calls to create a Tcl interpreter, and then evaluate a script file or perform other operations supported by the Tcl API. Then just link against '`libtcl.a`' and off you go.

Having done this, you have essentially created a shell. That is, your program can now execute shell scripts in the Tcl language. Tcl provides support for basic control flow, variable substitution file i/o and subroutines. In addition to the builtin Tcl commands, you can define your own subroutines as Tcl procecdures which effectively become new keywords.

But the real power of this approach is to add new commands to the interpretter which are realized by compiled C code in your application. Tcl provides a straightforward API call which allows you to register a function in your code to be called whenever the interpretter comes across a specific keyword of your choosing in the shell scripts it executes.

This facility allows you with tremendous ease, to endow your application with a powerful, robust and full featured macro language, trivially extend that macro language with new keywords which trigger execution of compiled application specific commands, and thereby raise the level of interaction with your code to one of essentially shell programming via script editing.

5.1.3 Acquiring Tcl

There are several important sources of info and code for Tcl. Definitely get the book mentioned above. The Tcl and Tk toolkits are distributed by anonymous ftp at '`sprite.berkeley.edu:/tcl`'. There are several files in there corresponding to Tcl, Tk, and various forms of documentation. At the time of this writing, the current versions of Tcl and Tk are 7.3 and 3.6 respectively. Retrieve those files, and install using the instructions provided therein.

The other major anonymou ftp site for Tcl is '`harbor.ecn.purdue.edu:/pub/tcl`'. Harbor contains a mirror of '`sprite`' as well as innumerable extensions, Tcl/Tk packages, tutorials, documentation, etc. The level of excitement in the Tcl community is extraordinarily high, and this is reflected by the great plethora of available, high quality, packages and extensions available for use with Tcl and Tk. Explore–there is definitely something for everyone.

Additionally there is a newsgroup, '`comp.lang.tcl`' which is well read, and an excellent place for people to get oriented, find help, etc. Highly recommended.

In any event, in order to use the Tk and Tcl-DP drivers in PLplot, you will need at a miniumum, '`tcl7.3.tar.gz`', '`tk3.6.tar.gz`' and '`tcl-dp-3.1.tar.gz`'. Additionally, in order to use the extended WISH paradigm (described below) you will need '`itcl-1.5.tar.gz`'.

Because of these dependencies, the PLplot ftp archive site provides these files as well. You can retrieve them when you get PLplot.

However, you will quite likely find Tcl/Tk to be very addictive, and the great plethora of add-ons available at '**harbor**' wil undoubtedly attract no small amount of your attention. It has been our experience that all of these extensions fit together very well. You will find that there are large sectors of the Tcl user community which create so-called "MegaWishes" which combine many of the available extensions into a single, heavily embellished, shell interpretter. The benefits of this approach will become apparent as you gain experience with Tcl and Tk.

5.2 Introduction to Tk

As mentioned above, Tcl is designed to be extensible. The first and most basic Tcl extension is Tk, an X11 toolkit. Tk provides the same basic facilities that you may be familiar with from other X11 toolkits such as Athena and Motif, except that they are provided in the context of the Tcl language. There are C bindings too, but these are seldom needed–the vast majority of useful Tk applications can be coded using Tcl scripts.

If it has not become obvious already, it is worth noting at this point that Tcl is one example of a family of languages known generally as "Very High Level Languages", or VHLL's. Essentially a VHLL raises the level of programming to a very high level, allowing very short token streams to accomplish as much as would be required by many scores of the more primitive actions available in a basic HLL. consider, for example, the basic "Hello World!" application written in Tcl/Tk.

```
#!/usr/local/bin/wish -f

button .hello -text "Hello World!"  -command "destroy ."
pack .hello
```

That's it! That's all there is to it. If you have ever programmed X using a traditional toolkit such as Athena or Motif, you can appreciate how amazingly much more convenient this is. If not, you can either take our word for it that this is 20 times less code than you would need to use a standard toolkit, or you can go write the same program in one of the usual toolkits and see for yourself...

We cannot hope to provide a thorough introduction to Tk programming in this section. Instead, we will just say that immensely complex applications can be constructed merely by programming in exactly the way shown in the above script. By writing more complex scripts, and by utilizing the additional widggets provided by Tk, one can create beautiful, extensive user interfaces. Moreover, this can be done in a tiny fraction of the time it takes to do the same work in a conventional toolkit. Literally minutes versus days.

Tk provides widgets for labels, buttons, radio buttons, frames with or without borders, menubars, pull downs, toplevels, canvases, edit boxes, scroll bars, etc.

A look at the interface provided by the PLplot Tk and Tcl-DP drivers should help give you a better idea of what you can do with this paradigm. Also check out some of the contributed Tcl/Tk packages available at harbor. There are high quality Tk interfaces to a great many familiar unix utilites ranging from mail to info, to SQL, to news, etc. The list is endless and growing fast...

5.3 Introduction to [incr Tcl]

Another extremely powerful and popular extension to Tcl is [incr Tcl]. [incr Tcl] is to Tcl what C++ is to C. The analogy is very extensive. Itcl provides an object orientated extension to Tcl supporting clustering of procedures and data into what is called an `itcl_class`. An `itcl_class` can have methods as well as instance data. And they support inheritance. Essentially if you know how C++ relates to C, and if you know Tcl, then you understand the programming model provided by Itcl.

In particular, you can use Itcl to implement new widgets which are composed of more basic Tk widgets. A file selector is an example. Using Tk, one can build up a very nice file selector comprised of more basic Tk widgets such as entries, listboxes, scrollbars, etc.

But what if you need two file selectors? You have to do it all again. Or what if you need two different kinds of file selectors, you get to do it again and add some incremental code.

This is exactly the sort of thing object orientation is intended to assist. Using Itcl you can create an `itcl_class FileSelector` and then you can instantiate them freely as easily as:

```
FileSelector .fs1
.fs1 -dir . -find "*.cc"
```

and so forth.

These high level widgets composed of smaller Tk widgets, are known as "megawidgets". There is a developing subculture of the Tcl/Tk community for designing and implementing megawidgets, and [incr Tcl] is the most popular enabling technology.

In particular, it is the enabling technology which is employed for the construction of the PLplot Tcl extensions, described below.

5.4 PLplot Extensions to Tcl

Following the paradigm described above, PLplot provides extensions to Tcl as well, designed to allow the use of PLplot from Tcl/Tk programs. Essentially the idea here is to allow PLplot programmers to achieve two goals:

- To access PLplot facilties from their own extended WISH and/or Tcl/Tk user interface scripts.
- To have PLplot display its output in a window integrated directly into the rest of their Tcl/Tk interface.

For instance, prior to PLplot 5.0, if a programmer wanted to use PLplot in a Tcl/Tk application, the best he could manage was to call the PLplot C api from compiled C code, and get the output via the Xwin driver, which would display in it's own toplevel window. In other words, there was no integration, and the result was pretty sloppy.

With PLplot 5.0, there is now a supported Tcl interface to PLplot functionality. This is provided through a "family" of PLplot megawidgets implemented in [incr Tcl]. Using this interface, a programmer can get a PLplot window/widget into a Tk interface as easily as:

```
PLWin .plw
pack .plw
```

Actually, there's the update/init business–need to clear that up.

The `PLWin` class then mirrors much of the PLplot C API, so that a user can generate plots in the PLplot widget entirely from Tcl. This is demonstrated in the '`tk02`' demo,

5.5 Custom Extensions to Tcl

By this point, you should have a pretty decent understanding of the underlying philosophy of Tcl and Tk, and the whole concept of extensions, of which [incr Tcl] and PLplot are examples. These alone are enough to allow the rapid prototyping and development of powerful, flexible grpahical applications. Normally the programmer simply writes a shell script to be executed by the Tk windowing shell, **wish**. It is in vogue for each Tcl/Tk extension package to build it's own "extended WISH". Tcl-DP, for example builds **dpwish** which users can use to evaluate scripts combining Tcl-DP commands with standard Tcl/Tk. There are many examples of this, and indeed even PLplot's **plserver** program, described in an earlier chapter, could just as easily have been called **plwish**.

In any event, as exciting and useful as these standalone, extended windowing shells may be, they are ultimately only the beginning of what you can do. The real benefit of this approach is realized when you make your own "extended WISH", comprised of Tcl, Tk, any of the standard extensions you like, and finally embellished with a smattering of application specific extensions designed to support your own application domain. In this section we give a detailed introduction to the process of constructing your own WISH. After that, you're on your own...

5.5.1 WISH Construction

The standard way to make your own WISH, as supported by the Tcl/Tk system, is to take a boilerplate file, '`tkAppInit.c`', edit to reflect the Tcl/Tk extensions you will be requiring, add some commands to the interpretter, and link it all together.

Here for example is the important part of the '`tk02`' demo, extracted from the file '`xtk02.c`', which is effectively the extended WISH definition file for the '`tk02`' demo. Comments and other miscelany are omitted.

```
#include "tk.h"
#include "itcl.h"

/* ... */

int   myplotCmd        (ClientData, Tcl_Interp *, int, char **);
```

```
int
Tcl_AppInit(interp)
    Tcl_Interp *interp; /* Interpreter for application. */
{
int   plFrameCmd        (ClientData, Tcl_Interp *, int, char **);

    Tk_Window main;

    main = Tk_MainWindow(interp);

    /*
     * Call the init procedures for included packages.  Each call should
     * look like this:
     *
     * if (Mod_Init(interp) == TCL_ERROR) {
     *     return TCL_ERROR;
     * }
     *
     * where "Mod" is the name of the module.
     */

    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Tk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Itcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Pltk_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }

    /*
     * Call Tcl_CreateCommand for application-specific commands, if
     * they weren't already created by the init procedures called above.
     */

    Tcl_CreateCommand(interp, "myplot", myplotCmd,
                      (ClientData) main, (void (*)(ClientData)) NULL);
```

```
    /*
     * Specify a user-specific startup file to invoke if the
     * application is run interactively.  Typically the startup
     * file is "~/.apprc" where "app" is the name of the application.
     * If this line is deletedthen no user-specific startup file
     * will be run under any conditions.
     */

    tcl_RcFileName = "~/.wishrc";
    return TCL_OK;
}

/* ... myPlotCmd, etc ... */
```

The calls to `Tcl_Init()` and `Tk_Init()` are in every WISH. To make an extended WISH, you add calls to the initialization routines for any extension packages you want to use, in this [incr Tcl] (`Itcl_Init()`) and PLplot (`Pltk_Init()`). Finally you add keywords to the interpretter, associating them with functions in your code using `Tcl_CreateCommand()` as shown.

In particular, PLplot has a number of [incr Tcl] classes in its Tcl library. If you want to be able to use those in your WISH, you need to include the initialization of [incr Tcl].

5.5.2 WISH Linking

Having constructed your `Tcl_AppInit()` function, you now merely need to link this file with your own private files to provide the code for any functions you registered via `Tcl_CreateCommand()` (and any they depend on), against the Tcl, Tk and extension libraries you are using.

```
cc -c tkAppInit.c
cc -c mycommands.c
cc -o my_wish tkAppInit.o mycommands.o
          -lplplotftk -ltcl -ltk -litcl -lX11 -lm
```

Add any needed `-L` options as needed.

Voila! You have made a wish.

5.5.3 WISH Programming

Now you are ready to put the genie to work. The basic plan here is to write shell scripts which use your new application specific windowing shell as their interpreter, to imlement X Windows user interfaces to control and utilize the facilities made available in your extensions.

Effectively this just comes down to wrting Tcl/Tk code, embellished as appropriate with calls to the extension commands you registered. Additionally, since this wish includes the PLplot extensions, you can instantiate any of the PLplot family of [incr Tcl] classes, and invoke methods on those objects to effect the drawing of graphs. Similarly, you may have your extension commands (which are coded in C) call the PLplot C programmers API to draw into the widget. In this way you can have the best of both worlds. Use compiled C code when the computational demands require the speed of compiled code, or use Tcl

Part III

Reference

Chapter 6

The PLplot Applications Programming Interface

Documentation of each PLplot function.

Many new ones are still missing.

`pladv(sub)`

Advances to the next subpage if **sub=0**, performing a page advance if there are no remaining subpages on the current page. If subwindowing isn't being used, **pladv(0)** will always advance the page. If **sub>0**, PLplot switches to the specified subpage. Note that this allows you to overwrite a plot on the specified subpage; if this is not what you intended, use **plclr** followed by **plpage** to first advance the page. This routine is called automatically (with **sub=0**) by **plenv**, but if **plenv** is not used, **pladv** must be called after initializing PLplot but before defining the viewport.

sub (PLINT, input):
Specifies the subpage number (starting from 1 in the top left corner and increasing along the rows) to which to advance. Set to zero to advance to the next subpage.

`plbin(nbin, x, y, cen)`

Plots a histogram consisting of **n** bins. The value associated with the **i**'th bin is placed in **x[i]**, and the number of points in the bin is placed in **y[i]**. For proper operation, the values in **x[i]** must form a strictly increasing sequence. If **center** is false, **x[i]** is the left-hand edge of the **i'th** bin, and if **center** is true, the bin boundaries are placed midway between the values in the **x** array. Also see **plhist** for drawing histograms from unbinned data.

nbin (PLINT, input):
Number of bins (i.e., number of values in \$x\$ and \$y\$ arrays.)

`x` (PLFLT *, input):
Pointer to array containing values associated with bins. These must form a strictly increasing sequence.

`y` (PLFLT *, input):
Pointer to array containing number of points in bin. This is a PLFLT (instead of PLINT) array so as to allow histograms of probabilities, etc.

`cen` (PLINT, input):
Indicates whether the values in `x` represent the lower bin boundaries (`cen=0`) or whether the bin boundaries are to be midway between the `x` values (`cen=1`). If the values in `x` are equally spaced and `cen=1`, the values in `x` are the center values of the bins.

`plbox(xopt, xtick, nxsub, yopt, ytick, nysub)`

Draws a box around the currently defined viewport, and labels it with world coordinate values appropriate to the window. Thus `plbox` should only be called after defining both viewport and window. The character strings `xopt` and `yopt` specify how the box should be drawn as described below. If ticks and/or subticks are to be drawn for a particular axis, the tick intervals and number of subintervals may be specified explicitly, or they may be defaulted by setting the appropriate arguments to zero.

`xopt` (char *, input):
Pointer to character string specifying options for horizontal axis. The string can include any combination of the following letters (upper or lower case) in any order:

- `a`: Draws axis, X-axis is horizontal line `y=0`, and Y-axis is vertical line `x=0`.
- `b`: Draws bottom (X) or left (Y) edge of frame.
- `c`: Draws top (X) or right (Y) edge of frame.
- `g`: Draws a grid at the major tick interval.
- `i`: Inverts tick marks, so they are drawn outwards, rather than inwards.
- `l`: Labels axis logarithmically. This only affects the labels, not the data, and so it is necessary to compute the logarithms of data points before passing them to any of the drawing routines.
- `m`: Writes numeric labels at major tick intervals in the unconventional location (above box for X, right of box for Y).
- `n`: Writes numeric labels at major tick intervals in the conventional location (below box for X, left of box for Y).
- `s`: Enables subticks between major ticks, only valid if `t` is also specified.
- `t`: Draws major ticks.

`xtick` (PLFLT, input):
World coordinate interval between major ticks on the x axis. If it is set to zero, PLplot automatically generates a suitable tick interval.

`nxsub` (PLINT, input):
Number of subintervals between major x axis ticks for minor ticks. If it is set to zero, PLplot automatically generates a suitable minor tick interval.

`yopt` (char *, input):
Pointer to character string specifying options for vertical axis. The string can include any combination of the letters defined above for `xopt`, and in addition may contain:

- `v`: Write numeric labels for vertical axis parallel to the base of the graph, rather than parallel to the axis.

`ytick` (real, input):
World coordinate interval between major ticks on the y axis. If it is set to zero, PLplot automatically generates a suitable tick interval.

`nysub` (PLINT, input):
Number of subintervals between major y axis ticks for minor ticks. If it is set to zero, PLplot automatically generates a suitable minor tick interval.

`plbox3(xopt, xlabel, xtick, nxsub, yopt, ylabel, ytick, nysub,zopt, zlabel, ztick, nzsub)`

Draws axes, numeric and text labels for a three-dimensional surface plot. For a more complete description of three-dimensional plotting see 3.12 [3-d Plots], page 27.

`xopt` (char *, input):
Pointer to character string specifying options for the x axis. The string can include any combination of the following letters (upper or lower case) in any order:

- `b`: Draws axis at base, at height `z=zmin` where `zmin` is defined by call to `plw3d`. This character must be specified in order to use any of the other options.
- `i`: Inverts tick marks, so they are drawn downwards, rather than upwards.
- `l`: Labels axis logarithmically. This only affects the labels, not the data, and so it is necessary to compute the logarithms of data points before passing them to any of the drawing routines.
- `n`: Writes numeric labels at major tick intervals.
- `s`: Enables subticks between major ticks, only valid if `t` is also specified.
- `t`: Draws major ticks.
- `u`: If this is specified, the text label for the axis is written under the axis.

`xlabel` (char *, input):
Pointer to character string specifying text label for the x axis. It is only drawn if `u` is in the `xopt` string.

`xtick` (PLFLT, input):
World coordinate interval between major ticks on the x axis. If it is set to zero, PLplot automatically generates a suitable tick interval.

`nxsub` (PLINT, input):
Number of subintervals between major x axis ticks for minor ticks. If it is set to zero, PLplot automatically generates a suitable minor tick interval.

`yopt` (char *, input):
Pointer to character string specifying options for the y axis. The string is interpreted in the same way as `xopt`.

`ylabel` (char *, input):
Pointer to character string specifying text label for the y axis. It is only drawn if `u` is in the `yopt` string.

`ytick` (PLFLT, input):
World coordinate interval between major ticks on the y axis. If it is set to zero, PLplot automatically generates a suitable tick interval.

`nysub` (PLINT, input):
Number of subintervals between major y axis ticks for minor ticks. If it is set to zero, PLplot automatically generates a suitable minor tick interval.

`zopt` (char *, input):
Pointer to character string specifying options for the z axis. The string can include any combination of the following letters (upper or lower case) in any order:

- `b`: Draws z axis to the left of the surface plot.
- `c`: Draws z axis to the right of the surface plot.
- `d`: Draws grid lines parallel to the x-y plane behind the figure. These lines are not drawn until after `plot3d` or `plmesh` are called because of the need for hidden line removal.
- `i`: Inverts tick marks, so they are drawn away from the center.
- `l`: Labels axis logarithmically. This only affects the labels, not the data, and so it is necessary to compute the logarithms of data points before passing them to any of the drawing routines.

- `m`: Writes numeric labels at major tick intervals on the right-hand verical axis.
- `n`: Writes numeric labels at major tick intervals on the left-hand verical axis.
- `s`: Enables subticks between major ticks, only valid if `t` is also specified.
- `t`: Draws major ticks.
- `u`: If this is specified, the text label is written beside the left-hand axis.
- `v`: If this is specified, the text label is written beside the right-hand axis.

`zlabel` (char *, input):
Pointer to character string specifying text label for the z axis. It is only drawn if `u` or `v` are in the **`zopt`** string.

`ztick` (PLFLT, input):
World coordinate interval between major ticks on the z axis. If it is set to zero, PLplot automatically generates a suitable tick interval.

`nzsub` (PLINT, input):
Number of subintervals between major z axis ticks for minor ticks. If it is set to zero, PLplot automatically generates a suitable minor tick interval.

`plclr()`

Clears the graphics screen of an interactive device, or ejects a page on a plotter. See 6 [plpage], page 66 for more information.

`plcol(color)`

Sets the color for subsequent lines.

`color` (PLINT, input):
Integer representing the color. The defaults at present are (these may change):

1. black (default background)
2. red
3. yellow
4. green
5. aquamarine
6. pink
7. wheat
8. grey
9. brown

10. blue
11. BlueViolet
12. cyan
13. turquoise
14. magenta
15. salmon
16. white (default foreground)

`plcon0(z, nx, ny, kx, lx, ky, ly, clevel, nlevel)` (Fortran)

Draws a contour plot of the data in `z[nx][ny]`, using the `nlevel` contour levels specified by `clevel`. Only the region of the array from `kx` to `lx` and from `ky` to `ly` is plotted out. See 3.13 [Contour plots], page 29 for more information.

`z` (PLFLT **, input):
Pointer to a vectored two-dimensional array containing data to be contoured.

`nx, ny` (PLINT, input):
Physical dimensions of array `z`.

`kx, lx` (PLINT, input):
Range of `x` indicies to consider.

`ky, ly` (PLINT, input):
Range of `y` indicies to consider.

`clevel` (PLFLT *, input):
Pointer to array specifying levels at which to draw contours.

`nlevel` (PLINT, input):
Number of contour levels to draw.

NOTE: this function is intended for use from a Fortran caller only. The C user should instead call `plcont` using the built-in transformation function `pltr0` for the same capability.

`plcon1(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, xg, yg)` (Fortran)

Draws a contour plot of the data in `z[nx][ny]`, using the `nlevel` contour levels specified by `clevel`. Only the region of the array from `kx` to `lx` and from `ky` to `ly` is plotted out. The arrays `xg` and `yg` are used to specify the transformation between array indices and world coordinates. See 3.13 [Contour plots], page 29 for more information.

`z` (PLFLT **, input):
Pointer to a vectored two-dimensional array containing data to be contoured.

`nx, ny` (PLINT, input):
Physical dimensions of array `z`.

`kx, lx` (PLINT, input):
Range of `x` indicies to consider.

`ky, ly` (PLINT, input):
Range of `y` indicies to consider.

`clevel` (PLFLT *, input):
Pointer to array specifying levels at which to draw contours.

`nlevel` (PLINT, input):
Number of contour levels to draw.

`xg, yg` (PLFLT *, input):
Pointers to arrays which specify the transformation from array indices to world coordinates. These must be one-dimensional arrays, used for a transformation of the form: `tx$=$f(x), ty$=$f(y)`. Function values at locations between grid points are obtained via linear interpolation.

NOTE: this function is intended for use from a Fortran caller only. The C user should instead call `plconf` using the built-in transformation function `pltr1` for the same capability.

`plcon2(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, xg, yg)` (Fortran)

Draws a contour plot of the data in `z[nx][ny]`, using the `nlevel` contour levels specified by `clevel`. Only the region of the array from `kx` to `lx` and from `ky` to `ly` is plotted out. The arrays `xg` and `yg` are used to specify the transformation between array indices and world coordinates. See 3.13 [Contour plots], page 29 for more information.

`z` (PLFLT **, input):
Pointer to a vectored two-dimensional array containing data to be contoured.

`nx, ny` (PLINT, input):
Physical dimensions of array `z`.

kx, lx (PLINT, input):
Range of **x** indicies to consider.

ky, ly (PLINT, input):
Range of **y** indicies to consider.

clevel (PLFLT *, input):
Pointer to array specifying levels at which to draw contours.

nlevel (PLINT, input):
Number of contour levels to draw.

xg, yg (PLFLT *, input):
Pointers to arrays which specify the transformation from array indices to world coordinates. These must be two-dimensional arrays, used for a transformation of the form: **tx$=$f(x, y), ty$=$f(x, y)**. Function values at locations between grid points are obtained via linear interpolation.

NOTE: this function is intended for use from a Fortran caller only. The C user should instead call **plconf** using the built-in transformation function **pltr2** for the same capability.

plconf(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, pltr, xg, yg) (C)

Draws a contour plot of the data in **z[nx][ny]**, using the **nlevel** contour levels specified by **clevel**. Only the region of the array from **kx** to **lx** and from **ky** to **ly** is plotted out. A transformation routine **pltr** is used to map indicies within the array to the world coordinates, using the transformation arrays **xg** and **yg**. See 3.13 [Contour plots], page 29 for more information.

z (PLFLT **, input):
Pointer to a vectored two-dimensional array containing data to be contoured.

nx, ny (PLINT, input):
Physical dimensions of array **z**.

kx, lx (PLINT, input):
Range of **x** indicies to consider.

ky, ly (PLINT, input):
Range of **y** indicies to consider.

`clevel` (PLFLT *, input):
: Pointer to array specifying levels at which to draw contours.

`nlevel` (PLINT, input):
: Number of contour levels to draw.

`pltr` (void *, input):
: Pointer to function that defines transformation between indicies in array `z` and the world coordinates. The function should have the form

  `pltr(x, y, tx, ty, xg, yg, nx, ny)`

  `x, y` (PLFLT, input):
  : Specifies the position in the array through which the contour runs in terms of the array indicies.

  `tx, ty` (PLFLT *, output):
  : Pointers to the world coordinates corresponding to the point `(x, y)`.

  `xg, yg` (PLFLT *, input):
  : Pointers to transformation arrays.

  `nx, ny` (PLINT, input):
  : Array dimensions in `x` and `y`, respectively.

  `xg, yg` (PLFLT *, input):
  : Pointers to arrays which define the transformation from `(x, y)` to `(tx, ty)`, used as input to the transformation function `pltr`.

Two transformation functions are provided: `pltr1`, for one dimensional mappings (i.e. `tx = f(x), ty = f(y)`), and `pltr2`, for two dimensional mappings (i.e. `tx = f(x, y), ty = f(x, y)`). Function values at locations between grid points are obtained via linear interpolation.

NOTE: this routine cannot be called directly from Fortran. Use `plcon0`, `plcon1`, or `plcon2` instead.

`plcont(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, pltr)`	(C)
`plcont(z, nx, ny, kx, lx, ky, ly, clevel, nlevel, tr)`	(Fortran)

Draws a contour plot of the data in `z[nx][ny]`, using the `nlevel` contour levels specified by `clevel`. Only the region of the array from `kx` to `lx` and from `ky` to `ly` is plotted out. A transformation routine `pltr` is used to map indicies within the array to the world coordinates. See 3.13 [Contour plots], page 29 for more information.

`z` (PLFLT **, input):
Pointer to a vectored two-dimensional array containing data to be contoured.

`nx, ny` (PLINT, input):
Physical dimensions of array `z`.

`kx, lx` (PLINT, input):
Range of `x` indicies to consider.

`ky, ly` (PLINT, input):
Range of `y` indicies to consider.

`clevel` (PLFLT *, input):
Pointer to array specifying levels at which to draw contours.

`nlevel` (PLINT, input):
Number of contour levels to draw.

`pltr` (void *, input):
Pointer to function that defines transformation between indicies in array `z` and the world coordinates (C only). Two transformation functions are provided: `xform`, for simple linear mappings, and `pltr0`, if no mapping is required. `xform` is not actually compiled into the PLplot library due to the need for external data; instead you must include it as part of your program. The function should have the form:

`pltr(x, y, tx, ty)`

`x, y` (PLFLT, input):
Specifies the position in the array through which the contour runs in terms of the array indicies.

`tx, ty` (PLFLT *, output):
Pointers to the world coordinates corresponding to the point `(x, y)`.

`tr` (PLFLT *, input):
Pointer to 6-element array specifying coefficients to use in the `xform` transformation function (Fortran only). Since this is somewhat inflexible, the user is recommended to call either of `plcon0`, `plcon1`, or `plcon2` instead.

`plend()`

Ends a plotting session, tidies up all the output files, switches interactive devices back into text mode and frees up any memory that was allocated. Must be called before end of program.

`plend1()`

Ends a plotting session for the current output stream only. See **plsstrm** for more info.

`plenv(xmin, xmax, ymin, ymax, just, axis)`

Sets up plotter environment for simple graphs by calling **pladv** and setting up viewport and window to sensible default values. **plenv** leaves enough room around most graphs for axis labels and a title. When these defaults are not suitable, use the individual routines **plvspa**, **plvpor**, or **plvasp** for setting up the viewport, **plwind** for defining the window, and **plbox** for drawing the box.

xmin (PLFLT, input):
Value of x at left-hand edge of window.

xmax (PLFLT, input):
Value of x at right-hand edge of window.

ymin (PLFLT, input):
Value of y at bottom edge of window.

ymax (PLFLT, input):
Value of y at top edge of window.

just (PLINT, input):
If `just=0`, the x and y axes are scaled independently to use as much of the screen as possible, but if `just=1`, the scales of the x and y axes are made equal.

axis (PLINT, input):
Controls drawing of the box around the plot:

- `-2`: No box or annotation.
- `-1`: Draw box only.
- `0`: Draw box, labelled with coordinate values around edge.
- `1`: In addition to box and labels, draw the two axes `x=0` and `y=0`.
- `2`: As for **axis=1**, but also draw a grid at the major tick interval.
- `10`: Logarithmic x axis, linear y axis.
- `11`: Logarithmic x axis, linear y axis and draw line `y=0`.
- `20`: Linear x axis, logarithmic y axis.
- `21`: Linear x axis, logarithmic y axis and draw line `x=0`.
- `30`: Logarithmic x and y axes.

plerrx(n, xmin, xmax, y)

Draws a set of **n** horizontal error bars, the **i**'th error bar extending from **xmin[i]** to **xmax[i]** at y coordinate **y[i]**. The terminals of the error bar are of length equal to the minor tick length (settable using **plsmin**).

n (PLINT, input):
Number of error bars to draw.

xmin (PLFLT *, input):
Pointer to array with x coordinates of left-hand endpoint of error bars.

xmax (PLFLT *, input):
Pointer to array with x coordinates of right-hand endpoint of error bars.

y (PLFLT *, input):
Pointer to array with y coordinates of error bar.

plerry(n, x, ymin, ymax)

Draws a set of **n** vertical error bars, the **i**'th error bar extending from **ymin[i]** to **ymax[i]** at x coordinate **x[i]**. The terminals of the error bar are of length equal to the minor tick length (settable using **plsmin**).

n (PLINT, input):
Number of error bars to draw.

x (PLFLT *, input):
Pointer to array with x coordinates of error bars.

ymin (PLFLT *, input):
Pointer to array with y coordinates of lower endpoint of error bars.

ymax (PLFLT *, input):
Pointer to array with y coordinate of upper endpoint of error bar.

plexit(message)

This routine is called in case an error is encountered during execution of a PLplot routine. It prints the error message, tries to release allocated resources, calls **pl_exit** and then exits. If cleanup needs to be done in the driver program then the user may want to supply his/her own version of **pl_exit**.

message (char *, input):
Error message.

plfill(n, x, y)

Fills the polygon defined by the **n** points **(x[i], y[i])** using the pattern defined by **plpsty** or **plpat**. The routine will automatically close the polygon between the last and first vertices. If multiple closed polygons are passed in **x** and **y** then **plfill** will fill in between them.

n (PLINT, input):
Number of vertices in polygon.

x (PLFLT *, input):
Pointer to array with x coordinates of vertices.

y (PLFLT *, input):
Pointer to array with y coordinates of vertices.

plfont(font)

Sets the default character font for subsequent character drawing. Also affects symbols produced by **plpoin**.

This routine has no effect unless the extended character set is loaded (see **plfontld**).

`font` (PLINT, input):
Specifies the font:

- `1`: Normal font (simplest and fastest)
- `2`: Roman font
- `3`: Italic font
- `4`: Script font

`plfontld(set)`

Sets the character set to use for subsequent character drawing. May be called before calling initializing PLplot.

`set` (PLINT, input):
Specifies the character set to load:

- `0`: Standard character set
- `1`: Extended character set

`plgfam(fam, num, bmax)`

Gets information about current family file, if familying is enabled. See Section **??** for more information.

`fam` (PLINT *, output):
Pointer to variable with the boolean family flag value. If nonzero, familying is enabled.

`num` (PLINT *, output):
Pointer to variable with the current family file number.

`bmax` (PLINT *, output):
Pointer to variable with the maximum file size (in bytes) for a family file.

`plgfile(file)`

Gets the current output file handle, if applicable.

`file` (FILE *, output):
File pointer to current output file.

NOTE: cannot be called from Fortran.

`plgfnam(fnam)`

Gets the current output file name, if applicable.

fnam (char *, output):
Pointer to file name string.

`plgpage(xp, yp, xleng, yleng, xoff, yoff)`

Gets the current page configuration.

xp (PLINT *, output):
Pointer to number of pixels in x.

yp (PLINT *; output):
Pointer to number of pixels in y.

xleng (PLINT *, output):
Pointer to x page length value.

yleng (PLINT *, output):
Pointer to y page length value.

xoff (PLINT *, output):
Pointer to x page offset.

yoff (PLINT *, output):
Pointer to y page offset.

`plgra()`

Sets an interactive device to graphics mode, used in conjunction with **pltext** to allow graphics and text to be interspersed. On a device which supports separate text and graphics windows, this command causes control to be switched to the graphics window. If already in graphics mode, this command is ignored. It is also ignored on devices which only support a single window or use a different method for shifting focus. See also **pltext**.

`plgspa(xmin, xmax, ymin, ymax)`

Gets the size of the current subpage in millimeters measured from the bottom left hand corner of the output device page or screen. Can be used in conjunction with **plsvpa** for setting the size of a viewport in absolute coordinates (millimeters).

xmin (PLFLT *, output):
Pointer to variable with position of left hand edge of subpage in millimeters.

xmax (PLFLT *, output):
Pointer to variable with position of right hand edge of subpage in millimeters.

ymin (PLFLT *, output):
Pointer to variable with position of bottom edge of subpage in millimeters.

ymax (PLFLT *, output):
Pointer to variable with position of top edge of subpage in millimeters.

`plgstrm(strm)`

Gets the number of the current output stream. See also **plsstrm**.

strm (PLINT *, output):
Pointer to current stream value.

`plgxax(digmax, digits)`

Returns current values of the **digmax** and **digits** flags for the x axis. **digits** is updated after the plot is drawn, so so this routine should only be called *after* the call to **plbox** (or **plbox3**) is complete. See 3.6 [Annotation], page 22 for more information.

digmax (PLINT *, output):
Pointer to variable with the maximum number of digits. If nonzero, the printed label will be switched to a floating point representation when the number of digits exceeds **digmax**.

digits (PLINT *, output):
Pointer to variable with the actual number of digits for the numeric labels (x axis) from the last plot.

`plgyax(digmax, digits)`

Identical to **plgxax**, except that arguments are flags for y axis. See the description of **plgxax** for more detail.

`plgzax(digmax, digits)`

Identical to **plgxax**, except that arguments are flags for z axis. See the description of **plgxax** for more detail.

`plhist(n, data, datmin, datmax, nbin, oldwin)`

Plots a histogram from **n** data points stored in the array **data**. This routine bins the data into **nbin** bins equally spaced between **datmin** and **datmax**, and calls **plbin** to draw the resulting histogram. Parameter **oldwin** allows the histogram either to be plotted in an existing window or causes **plhist** to call **plenv** with suitable limits before plotting the histogram.

n (PLINT, input):
Number of data points.

data (PLFLT *, input):
Pointer to array with values of the **n** data points.

datmin (PLFLT, input):
Left-hand edge of lowest-valued bin.

datmax (PLFLT, input):
Right-hand edge of highest-valued bin.

nbin (PLINT, input):
Number of (equal-sized) bins into which to divide the interval **xmin** to **xmax**.

oldwin (PLINT, input):
If one, the histogram is plotted in the currently-defined window, and if zero, **plenv** is called automatically before plotting.

`plinit()`

Initializing the plotting package. The program prompts for the device keyword or number of the desired output device. Hitting a RETURN in response to the prompt is the same as selecting the first device. **plinit** will issue no prompt if either the device was specified previously (via command line flag or the **plsdev** function), or if only one device is enabled when PLplot is installed. If subpages have been specified, the output device is divided into **nx** by **ny** sub-pages, each of which may be used independently. If **plinit** is called again during a program, the previously opened file will be closed. The subroutine **pladv** is used to advance from one subpage to the next.

`pljoin(x1, y1, x2, y2)`

Joins the point **(x1, y1)** to **(x2, y2)**.

x1 (PLFLT, input):
x coordinate of first point.

y1 (PLFLT, input):
y coordinate of first point.

x2 (PLFLT, input):
x coordinate of second point.

y2 (PLFLT, input):
y coordinate of second point.

`pllab(xlabel, ylabel, tlabel)`

Routine for writing simple labels. Use **plmtex** for more complex labels.

xlabel (char *, input):
Label for horizontal axis.

ylabel (char *, input):
Label for vertical axis.

tlabel (char *, input):
Title of graph.

`plline(n, x, y)`

Draws `n-1` line segments joining points `(x[i], y[i])`.

`n` (PLINT, input):
Number of points to join.

`x` (PLFLT *, input):
Pointer to array with x coordinates of points.

`y` (PLFLT *, input):
Pointer to array with y coordinates of points.

`pllsty(n)`

This sets the line style according to one of eight predefined patterns (also see `plstyl`).

`n` (PLINT, input):
Integer value between 1 and 8.

`plmesh(x, y, z, nx, ny, opt)` (C)

`plmesh(x, y, z, nx, ny, opt, mx)` (Fortran)

Plots a surface mesh within the environment set up by `plw3d`. The surface is defined by the two-dimensional array `z[][]`, the point `z[i][j]` being the value of the function at `(x[i], y[j])`. Note that the points in arrays `x` and `y` do not need to be equally spaced, but must be stored in ascending order. The parameter `opt` controls the way in which the surface is displayed. For further details see 3.12 [3-d Plots], page 27.

`x` (PLFLT *, input):
Pointer to set of x coordinate values at which the function is evaluated.

`y` (PLFLT *, input):
Pointer to set of y coordinate values at which the function is evaluated.

`z` (PLFLT **, input):
Pointer to a vectored two-dimensional array with set of function values.

`nx` (PLINT, input):
Number of `x` values at which function is evaluated.

`ny` (PLINT, input):
Number of `y` values at which function is evaluated.

`opt` (PLINT, input):
Determines the way in which the surface is represented:

- `1`: Lines are drawn showing `z` as a function of `x` for each value of `y[j]`.
- `2`: Lines are drawn showing `z` as a function of `y` for each value of `x[i]`.
- `3`: Network of lines is drawn connecting points at which function is defined.

`mx` (PLINT, input):
Length of array in x direction, for plotting subarrays (Fortran only).

`plmtex(side, disp, pos, just, text)`

Writes text at a specified position relative to the viewport boundaries. Text may be written inside or outside the viewport, but is clipped at the subpage boundaries. The reference point of a string lies along a line passing through the string at half the height of a capital letter. The position of the reference point along this line is determined by `just`, and the position of the reference point relative to the viewport is set by `disp` and `pos`.

`side` (char *, input):
Specifies the side of the viewport along which the text is to be written. The string must be one of:

- `b`: Bottom of viewport.
- `l`: Left of viewport, text written parallel to edge.
- `lv`: Left of viewport, text written at right angles to edge.
- `r`: Right of viewport, text written parallel to edge.
- `rv`: Right of viewport, text written at right angles to edge.
- `t`: Top of viewport.

`disp` (PLFLT, input):
Position of the reference point of string, measured outwards from the specified viewport edge in units of the current character height. Use negative `disp` to write within the viewport.

`pos` (PLFLT, input):
Position of the reference point of string along the specified edge, expressed as a fraction of the length of the edge.

`just` (PLFLT, input):
Specifies the position of the string relative to its reference point. If `just=0`, the reference point is at the left and if `just=1`, it is at the right of the string. Other values of `just` give intermediate justifications.

`text` (char *, input):
The string to be written out.

`plot3d(x, y, z, nx, ny, opt, side)` (C)

`plot3d(x, y, z, nx, ny, opt, side, mx)` (Fortran)

Plots a three dimensional surface plot within the environment set up by `plw3d`. The surface is defined by the two-dimensional array `z[][]`, the point `z[i][j]` being the value of the function at `(x[i], y[j])`. Note that the points in arrays `x` and `y` do not need to be equally spaced, but must be stored in ascending order. The parameter `opt` controls the way in which the surface is displayed. For further details see 3.12 [3-d Plots], page 27.

`x` (PLFLT *, input):
Pointer to set of x coordinate values at which the function is evaluated.

`y` (PLFLT *, input):
Pointer to set of y coordinate values at which the function is evaluated.

`z` (PLFLT **, input):
Pointer to a vectored two-dimensional array with set of function values.

`nx` (PLINT, input):
Number of `x` values at which function is evaluated.

`ny` (PLINT, input):
Number of `y` values at which function is evaluated.

`opt` (PLINT, input):
Determines the way in which the surface is represented:

- `1`: Lines are drawn showing `z` as a function of `x` for each value of `y[j]`.
- `2`: Lines are drawn showing `z` as a function of `y` for each value of `x[i]`.
- `3`: Network of lines is drawn connecting points at which function is defined.

`side` (PLINT, input):
Flag to indicate whether or not "sides" should be draw on the figure. If `side=0` no sides are drawn, otherwise the sides are drawn.

`mx` (PLINT, input):
Length of array in x direction, for plotting subarrays (Fortran only).

`plpage()`

Begins a new page. For a file driver, the output file is opened if necessary. Advancing the page via **plclr** and **plpage** is useful when a page break is desired at a particular point when plotting to subpages. Another use for **plclr** and **plpage** is when plotting pages to different files, since you can manually set the file name (or file handle) by calling **plsfnam** or **plsfile** after the call to **plclr** (in fact some drivers may only support a single page per file, making this a necessity). One way to handle this case automatically is to page advance via **pladv**, but enable familying (see **plsfam**) with a small byte per file limit so that a new family member file will be created on each page break.

Except for one of these circumstances, it is best to advance the page or subpage simply by the call `pladv(0)`.

`plpat(nlin, inc, del)`

Sets the area fill pattern. The pattern consists of 1 or 2 sets of parallel lines with specified inclinations and spacings. The arguments to this routine are the number of sets to use (1 or 2) followed by two pointers to integer arrays (of 1 or 2 elements) specifying the inclinations in tenths of a degree and the spacing in micrometers. (also see **plpsty**)

`nlin` (PLINT, input):
Number of sets of lines making up the pattern, either 1 or 2.

`inc` (PLINT *, input):
Pointer to array with `nlin` elements. Specifies the line inclination in tenths of a degree. (Should be between -900 and 900).

`del` (PLINT *, input):
Pointer to array with `nlin` elements. Specify the spacing in micrometers between the lines making up the pattern.

`plpoin(n, x, y, code)`

Marks out a set of `n` points at positions `(x(i), y(i))`, using the symbol defined by `code`. If `code` is between 32 and 127, the symbol is simply the printable ASCII character in the default font.

`n` (PLINT, input):
Number of points to be marked.

`x` (PLFLT *, input):
Pointer to array with set of x coordinate values for the points.

`y` (PLFLT *, input):
Pointer to array with set of y coordinate values for the points.

`code` (PLINT, input):
Code number for the symbol to be plotted.

`plprec(set, prec)`

Sets the number of places after the decimal point in numeric labels.

`set` (PLINT, input):
If `set` is equal to 0 then PLplot automatically determines the number of places to use after the decimal point in numeric labels (like those used to label axes). If `set` is 1 then `prec` sets the number of places.

`prec` (PLINT, input):
The number of characters to draw after the decimal point in numeric labels.

`plpsty(n)`

Select one of eight predefined area fill patterns to use (also see `plpat`).

`n` (PLINT *, input):
The desired pattern.

`plptex(x, y, dx, dy, just, text)`

Writes text at a specified position and inclination within the viewport. Text is clipped at the viewport boundaries. The reference point of a string lies along a line passing through the string at half the height of a capital letter. The position of the reference point along this line is determined by `just`, the reference point is placed at world coordinates `(x, y)` within the viewport. The inclination of the string is specified in terms of differences of world coordinates making it easy to write text parallel to a line in a graph.

`x` (PLFLT, input):
x coordinate of reference point of string.

`y` (PLFLT, input):
y coordinate of reference point of string.

`dx` (PLFLT, input):
Together with `dy`, this specifies the inclination of the string. The baseline of the string is parallel to a line joining `(x, y)` to `(x+dx, y+dy)`.

`dy` (PLFLT, input):
Together with `dx`, this specifies the inclination of the string.

`just` (PLFLT, input):
Specifies the position of the string relative to its reference point. If `just=0`, the reference point is at the left and if `just=1`, it is at the right of the string. Other values of `just` give intermediate justifications.

`text` (char *, input):
The string to be written out.

`plschr(def, scale)`

This sets up the size of all subsequent characters drawn. The actual height of a character is the product of the default character size and a scaling factor.

def (PLFLT, input):
The default height of a character in millimeters, should be set to zero if the default height is to remain unchanged.

scale (PLFLT, input):
Scale factor to be applied to default to get actual character height.

plsfam(fam, num, bmax)

Sets variables dealing with output file familying. Does nothing if familying not supported by the driver. This routine, if used, must be called before initializing PLplot. See Section **??** for more information.

fam (PLINT, input):
Family flag (boolean). If nonzero, familying is enabled.

num (PLINT, input):
Current family file number.

bmax (PLINT, input):
Maximum file size (in bytes) for a family file.

plsfile(file)

Sets the current output file handle, if applicable. If the file has has not been previously opened and is required by the driver, the user will be prompted for the file name. This routine, if used, must be called before initializing PLplot.

file (FILE *, input):
File pointer.

NOTE: cannot be called from Fortran.

plsfnam(fnam)

Sets the current output file name, if applicable. If the file name has not been specified and is required by the driver, the user will be prompted for it. If using the X-windows output driver, this sets the display name. This routine, if used, must be called before initializing PLplot.

fnam (char *, input):
Pointer to file name string.

`plsmaj(def, scale)`

This sets up the length of the major ticks. The actual length is the product of the default length and a scaling factor as for character height.

def (PLFLT, input):
The default length of a major tick in millimeters, should be set to zero if the default length is to remain unchanged.

scale (PLFLT, input):
Scale factor to be applied to default to get actual tick length.

`plsmin(def, scale)`

This sets up the length of the minor ticks and the length of the terminals on error bars. The actual length is the product of the default length and a scaling factor as for character height.

def (PLFLT, input):
The default length of a minor tick in millimeters, should be set to zero if the default length is to remain unchanged.

scale (PLFLT, input):
Scale factor to be applied to default to get actual tick length.

`plsori(ori)`

Sets the current orientation. If `ori` is equal to zero (default) then landscape is used (x axis is parallel to the longest edge of the page), otherwise portrait is used. This option is not supported by all output drivers (in particular, most interactive screen drivers ignore the orientation). This routine, if used, must be called before initializing PLplot.

ori (PLINT, input):
Orientation value.

`plspage(xp, yp, xleng, yleng, xoff, yoff)`

Sets the page configuration (optional). Not all parameters recognized by all drivers. The X-window driver uses the length and offset parameters to determine the window size and location. This routine, if used, must be called before initializing PLplot.

xp (PLINT *, output):
Number of pixels, x.

yp (PLINT *; output):
Number of pixels, y.

xleng (PLINT *, output):
Page length, x.

yleng (PLINT *, output):
Page length, y.

xoff (PLINT *, output):
Page offset, x.

yoff (PLINT *, output):
Page offset, y.

`plsstrm(strm)`

Sets the number of the current output stream. The stream number defaults to 0 unless changed by this routine. The first use of this routine must be followed by a call initializing PLplot (e.g. **plstar**).

strm (PLINT *, output):
Pointer to current stream number.

`plssym(def, scale)`

This sets up the size of all subsequent symbols drawn by **plpoin** and **plsym**. The actual height of a symbol is the product of the default symbol size and a scaling factor as for the character height.

def (PLFLT, input):
The default height of a symbol in millimeters, should be set to zero if the default height is to remain unchanged.

scale (PLFLT, input):
Scale factor to be applied to default to get actual symbol height.

plstar(nx, ny)

Initializing the plotting package. The program prompts for the device keyword or number of the desired output device. Hitting a RETURN in response to the prompt is the same as selecting the first device. If only one device is enabled when PLplot is installed, **plstar** will issue no prompt. The output device is divided into **nx** by **ny** sub-pages, each of which may be used independently. The subroutine **pladv** is used to advance from one subpage to the next.

nx (PLINT, input):
Number of subpages to divide output page in the horizontal direction.

ny (PLINT, input):
Number of subpages to divide output page in the vertical direction.

plstart(device, nx, ny)

Alternative to **plstar** for initializing the plotting package. The device name keyword for the desired output device must be supplied as an argument. The device keywords are the same as those printed out by **plstar**. If the requested device is not available, or if the input string is empty or begins with "?", the prompted startup of **plstar** is used. This routine also divides the output device into **nx** by **ny** sub-pages, each of which may be used independently. The subroutine **pladv** is used to advance from one subpage to the next.

device (char *, input):
Device name (keyword) of the required output device. If NULL or if the first character is a "?", the normal (prompted) startup is used.

nx (PLINT, input):
Number of subpages to divide output page in the horizontal direction.

`ny` (PLINT, input):
Number of subpages to divide output page in the vertical direction.

`plstyl(nels, mark, space)`

This sets up the line style for all lines subsequently drawn. A line consists of segments in which the pen is alternately down and up. The lengths of these segments are passed in the arrays `mark` and `space` respectively. The number of mark-space pairs is specified by `nels`. In order to return the line style to the default continuous line, `plstyl` should be called with `nels=0`.(also see `pllsty`)

`nels` (PLINT, input):
The number of `mark` and `space` elements in a line. Thus a simple broken line can be obtained by setting `nels=1`. A continuous line is specified by setting `nels=0`.

`mark` (PLINT *, input):
Pointer to array with the lengths of the segments during which the pen is down, measured in micrometers.

`space` (PLINT *, input):
Pointer to array with the lengths of the segments during which the pen is up, measured in micrometers.

`plsvpa(xmin, xmax, ymin, ymax)`

Alternate routine to `plvpor` for setting up the viewport. This routine should be used only if the viewport is required to have a definite size in millimeters. The routine `plgspa` is useful for finding out the size of the current subpage.

`xmin` (PLFLT, input):
The distance of the left-hand edge of the viewport from the left-hand edge of the subpage in millimeters.

`xmax` (PLFLT, input):
The distance of the right-hand edge of the viewport from the left-hand edge of the subpage in millimeters.

`ymin` (PLFLT, input):
The distance of the bottom edge of the viewport from the bottom edge of the subpage in millimeters.

ymax (PLFLT, input):
The distance of the top edge of the viewport from the top edge of the subpage in millimeters.

`plsxax(digmax, digits)`

Returns current values of the **digmax** and **digits** flags for the x axis. **digits** is updated after the plot is drawn, so so this routine should only be called *after* the call to **plbox** (or **plbox3**) is complete. See 3.6 [Annotation], page 22 for more information.

digmax (PLINT, output):
Variable to set the maximum number of digits. If nonzero, the printed label will be switched to a floating point representation when the number of digits exceeds **digmax**.

digits (PLINT, input):
Field digits value. Currently, changing its value here has no effect since it is set only by **plbox** or **plbox3**. However, the user may obtain its value after a call to **plbox** by calling **plgxax**.

`plsyax(digmax, digits)`

Identical to **plsxax**, except that arguments are flags for y axis. See the description of **plsxax** for more detail.

`plsym(n, x, y, code)`

Marks out a set of **n** points at positions `(x[i], y[i])`, using the symbol defined by **code**. The code is interpreted as an index in the Hershey font tables.

n (PLINT, input):
Number of points to be marked.

x (PLFLT *, input):
Pointer to array with set of x coordinate values for the points.

y (PLFLT *, input):
Pointer to array with set of y coordinate values for the points.

code (PLINT, input):
Code number for the symbol to be plotted.

plszax(digmax, digits)

Identical to **plsxax**, except that arguments are flags for z axis. See the description of **plsxax** for more detail.

pltext()

Sets an interactive device to text mode, used in conjunction with **plgra** to allow graphics and text to be interspersed. On a device which supports separate text and graphics windows, this command causes control to be switched to the text window. This can be useful for printing diagnostic messages or getting user input, which would otherwise interfere with the plots. The user *must* switch back to the graphics window before issuing plot commands, as the text (or console) device will probably become quite confused otherwise. If already in text mode, this command is ignored. It is also ignored on devices which only support a single window or use a different method for shifting focus (also see 6 [plgra], page 59).

plvasp(aspect)

Sets the viewport so that the ratio of the length of the y axis to that of the x axis is equal to **aspect**.

aspect (PLFLT, input):
Ratio of length of y axis to length of x axis.

plvpas(xmin, xmax, ymin, ymax, aspect)

Device-independent routine for setting up the viewport. The viewport is chosen to be the largest with the given aspect ratio that fits within the specified region (in terms of normalized subpage coordinates). This routine is functionally equivalent to **plvpor** when a "natural" aspect ratio (0.0) is chosen. Unlike **plvasp**, this routine reserves no extra space at the edges for labels.

xmin (PLFLT, input):
The normalized subpage coordinate of the left-hand edge of the viewport.

xmax (PLFLT, input):
The normalized subpage coordinate of the right-hand edge of the viewport.

ymin (PLFLT, input):
The normalized subpage coordinate of the bottom edge of the viewport.

ymax (PLFLT, input):
The normalized subpage coordinate of the top edge of the viewport.

aspect (PLFLT, input):
Ratio of length of y axis to length of x axis.

`plvpor(xmin, xmax, ymin, ymax)`

Device-independent routine for setting up the viewport. This defines the viewport in terms of normalized subpage coordinates which run from 0.0 to 1.0 (left to right and bottom to top) along each edge of the current subpage. Use the alternate routine **plsvpa** in order to create a viewport of a definite size.

xmin (PLFLT, input):
The normalized subpage coordinate of the left-hand edge of the viewport.

xmax (PLFLT, input):
The normalized subpage coordinate of the right-hand edge of the viewport.

ymin (PLFLT, input):
The normalized subpage coordinate of the bottom edge of the viewport.

ymax (PLFLT, input):
The normalized subpage coordinate of the top edge of the viewport.

`plvsta()`

Sets up a standard viewport, leaving a left-hand margin of seven character heights, and four character heights around the other three sides.

`plw3d(basex, basey, height, xmin, xmax, ymin, ymax, zmin, zmax, alt, az)`

Sets up a window for a three-dimensional surface plot within the currently defined two-dimensional window. The enclosing box for the surface plot defined by **xmin**, **xmax**, **ymin**, **ymax**, **zmin** and **zmax** in user-coordinate space is mapped into a box of world coordinate size **basex** by **basey** by **height** so that **xmin** maps to **-basex/2**, **xmax** maps to **basex/2**, **ymin** maps to **-basey/2**, **ymax** maps to **basey/2**, **zmin** maps to **0** and **zmax** maps to **height**. The resulting world-coordinate box is then viewed by an observer at altitude **alt** and azimuth **az**. This routine must be called before **plbox3** or **plot3d**. For a more complete description of three-dimensional plotting see 3.12 [3-d Plots], page 27.

basex (PLFLT, input):
The x coordinate size of the world-coordinate box.

basey (PLFLT, input):
The y coordinate size of the world-coordinate box.

height (PLFLT, input):
The z coordinate size of the world-coordinate box.

xmin (PLFLT, input):
The minimum user x coordinate value.

xmax (PLFLT, input):
The maximum user x coordinate value.

ymin (PLFLT, input):
The minimum user y coordinate value.

ymax (PLFLT, input):
The maximum user y coordinate value.

zmin (PLFLT, input):
The minimum user z coordinate value.

zmax (PLFLT, input):
The maximum user z coordinate value.

alt (PLFLT, input):
The viewing altitude in degrees above the xy plane.

az (PLFLT, input):
The viewing azimuth in degrees. When **az=0**, the observer is looking face onto the zx plane, and as **az** is increased, the observer moves clockwise around the box when viewed from above the xy plane.

plwid(width)

Sets the pen width.

`width` (PLINT, input):
The desired pen width. The pen width must be between 1 and a device dependent maximum value.

`plwind(xmin, xmax, ymin, ymax)`

Sets up the world coordinates of the edges of the viewport.

`xmin` (PLFLT, input):
The world x coordinate of the left-hand edge of the viewport.

`xmax` (PLFLT, input):
The world x coordinate of the right-hand edge of the viewport.

`ymin` (PLFLT, input):
The world y coordinate of the bottom edge of the viewport.

`ymax` (PLFLT, input):
The world y coordinate of the top edge of the viewport.

Part IV

Closure

This is the bibliography.