The PLplot Plotting Library

Programmer's Reference Manual

Maurice J. LeBrun

Geoff Furnish
University of Texas at Austin
Institute for Fusion Studies

The PLplot Plotting Library: Programmer's Reference Manual

by Maurice J. LeBrun and Geoff Furnish

Release version: 5.2.1.cvs.20040104

Release date: 2004-01-04

Table of Contents

List of Tables

I. Introduction

Chapter 1. Introduction

The PLplot Plotting Library

PLplot is a library of C functions that are useful for making scientific plots from programs written in C, C++, Fortran, Octave, Python, and Tcl/Tk. The PLplot project is being developed by a world-wide team who interact via the facilities provided by SourceForge (http://sourceforge.net/projects/plplot)

The PLplot library can be used to create standard x-y plots, semi-log plots, log-log plots, contour plots, 3D plots, shade (gray-scale and color) 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 and file formats are supported including a metafile format which can be subsequently rendered to any device/file. New devices and file formats can be easily added by writing a driver routine. For example, we have recently added PNG and JPEG file drivers, and a GNOME interactive driver is being developed.

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 traveled 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 modeled 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.

Some of the improvements in PLplot 5.0 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 labeling 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 5.0 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: Unix/Linux, OS/2, Mac, MS-DOS, and Win9x. The Unix/Linux version is the best supported of these possibilities. The PLplot package is freely distributable, but *not* in the public domain. See the Section called *Copyrights* 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 the general PLplot mailing list, plplot-general@lists.sourceforge.net.

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 is a SourceForge project and may be obtained by the usual SourceForge file release and anonymous cvs access that is made available from links at http://sourceforge.net/projects/plplot.

Installing and Using the PLplot Library

The installation procedure is by necessity system specific; installation notes for each system are provided in Chapter 18. 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 C, C++, and Fortran are included as a guide. Plots generated from the example programs are shown here[3].

You will then need to compile your program and link it with the PLplot library(s). See Chapter 18 for more details).

You can also use Tcl/Tk scripts or Python scripts to generate plots using the PLplot libraries. Examples of these possibilities are also included as a guide.

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 accommodated by providing lower-level routines which change the

system defaults. The manual first describes the overall process of producing a graph using the high-level routines (see the Section called *Plotting a Simple Graph* in Chapter 2). For a discussion of the underlying concepts of the plotting process and an introduction to some of the more complex routines (see Chapter 3). An alphabetical list of the user-accessible PLplot functions with detailed descriptions is given in the reference section of the manual (see Chapter 15).

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 ref ap:lang ; 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 ref ap:lang 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 ref ap:dev , along with description of the device driver--PLplot interface, metafile output, family files, and vt100/tek4010 emulators. In Appendix ref ap:sys 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).

Copyrights

The PLplot package may be distributed under the following terms:

```
This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Library General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Library General Public License for more details.

You should have received a copy of the GNU Library General Public
License along with this library; if not, write to the Free
Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

The text of this license is given in the file COPYING.LIB in the distribution directory. Exceptions are noted below.

The intent behind distributing PLplot under the LGPL is to ensure that it continues to evolve in a positive way, while remaining freely distributable. The package is considered a library even though there are associated programs, such as plrender, pltek, plserver, and pltcl. The ties between these programs and the library are so great that I consider them as part of the library, so distribution under the terms of the LGPL makes sense. Software developers are allowed and encouraged to use PLplot as an integral part of their product, even a commercial product. Under the conditions of the LGPL, however, the PLplot source code must remain freely available, including any modifications you make to it (if you distribute a program based on the modified library). Please read the full license for more info.

Additional Copyrights

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:

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 LGPL and whatever stated conditions accompany the copyright.

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:

The Institute for Fusion Studies, University of Texas at Austin

The Scientific and Technology Agency of Japan

Japan Atomic Energy Research Institute

Duke University

Universite de Nice

National Energy Research Supercomputer Center

Los Alamos National Labs

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

Tony Richardson: Creator of PLplot 2.6b, 3.0

Sam Paolucci (postscript driver)

Sam Paolucci (postscript driver)

Tom Rokicki (IFF driver and Amiga printer driver)

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

Notes

1. http://sourceforge.net/projects/plplot
2. http://sourceforge.net/projects/plplot
3. http://plplot.sf.net/examples/index.html

II. Programming

Chapter 2. Simple Use of PLplot

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.

Initializing PLplot

Before any actual plotting calls are made, a graphics program must call `plinit`, 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` allows you to set the device explicitly. The function `plsetopt` allows you to set any command-line option internally in your code. The function `plssub` 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`. If subpages are used, this can be used to advance to the next subpage or to a particular subpage.

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, labeled 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* = $\log_{10}(1) = 0.0$, and *xmax* = $\log_{10}(1000) = 3.0$.

For greater control over the size of the plots, axis labeling and tick intervals, more complex graphs should make use of the functions plvpor, plvasp, plvpas, plwind, plbox, and routines for manipulating axis labeling plgxax through plszax.

Labeling 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. For discussion of writing text in a plot see the Section called *Writing Text on a Graph*, and for more detailed discussion about label generation see the Section called *Writing Text on a Graph*.

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 Chapter 3 for discussion of finer control of plot generation.

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

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 the Section called *Setting Line Attributes* in Chapter 3 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.

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.

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.

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.

More Complex Graphs

Functions plbin and plhist are provided for drawing histograms, and functions plerrx and plerry draw error bars about specified points. There are lots more too (see Chapter 15).

Finishing Up

Before the end of the program, *always* call plend 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.

In Case of Error

If a fatal error is encountered during execution of a PLplot routine then plexit is called. This routine prints an error message, does resource recovery, and then exits. The user may specify an error handler via plsexit 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 advanced use of PLplot.

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` 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 [options]

PLplot options:
    -h                  Print out this message
    -v                  Print out the PLplot library version number
    -verbose            Be more verbose than usual
    -debug              Print debugging info (implies -verbose)
    -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)
    -freeaspect         Do not preserve aspect ratio on orientation swaps
    -width width        Sets pen width (1 <= width <= 10)
    -bg color           Background color (0=black, FFFFFF=white)
    -ncol0 n            Number of colors to allocate in cmap 0 (upper bound)
    -ncol1 n            Number of colors to allocate in cmap 1 (upper bound)
    -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.
```

Output Devices

PLplot supports a variety of output devices, via a set of device drivers. Each driver is required to emulate a small set of low-level graphics primitives such as initialization, line draw and page advance, as well as be completely independent of the PLplot package as a whole. Thus a driver may be very simple, as in the case of the many black and white file drivers (tektronix, etc.). More complicated and/or color systems require a bit more effort by the driver, with the most effort required by an output device with a graphical user interface, including menus for screen dumps, palette manipulation, and so forth. At present only the tk driver does the latter on Unix systems. At present we aren't pursuing a Macintosh development effort due to a lack of time and expertise, but will assist anyone wanting to volunteer for the job.

Note that if you always render to a PLplot metafile, you can always `plrender` them to new devices as they become available.

The list of available devices presented when starting PLplot (via `plstar`) is determined at compile time. When installing PLplot you may wish to exclude devices not available on your system in order to reduce screen clutter. To include a specified device, simply define the appropriate macro constant when building PLplot (see the installation instructions for your system).

The device drivers for PLplot terminal output at present are given in Table 3-1 while drivers for file output are given in Table 3-2. The driver for OS/2 PM is available separately. See the section on OS/2 in the Appendix for more details.

Table 3-1. PLplot Terminal Output Devices

Device	keyword	driver file
X-Window Screen	xwin	xwin.c
Tcl/Tk widget	tk	tk.c
Linux console VGA	vga	linuxvga.c
Xterm Window	xterm	tek.c
Tektronix Terminal (4010)	tekt	tek.c
Tektronix Terminal (4105/4107)	tek4107t	tek.c
MS-Kermit emulator	mskermit	tek.c
Versaterm vt100/tek emulator	versaterm	tek.c
VLT vt100/tek emulator	vlt	tek.c
Conex vt320/tek emulator	conex	tek.c
DG300 Terminal	dg300	dg300.c
NeXT display (unsupported)	nx	next.c
GNOME display	gnome	gnome.c

Table 3-2. PLplot File Output Devices

Device	keyword	driver file
PLplot Native Meta-File	plmeta	plmeta.c
Tektronix File (4010)	tekf	tek.c
Tektronix File (4105/4107)	tek4107f	tek.c
PostScript File (monochrome)	ps	ps.c
PostScript File (color)	psc	ps.c
XFig file	xfig	xfig.c
LaserJet IIp Bitmap File	ljiip	ljiip.c

Device	keyword	driver file
LaserJet II Bitmap File (150 dpi)	ljii	ljii.c
HP 7470 Plotter File (HPGL Cartridge Small Plotter)	hp7470	hpgl.c
HP 7580 Plotter File (Large Plotter)	hp7580	hpgl.c
HP Laser Jet, HPGL file	lj_hpgl	hpgl.c
Impress File	imp	impress.c
Portable bitmap file	pbm	pbm.c
Null device	null	null.c
JPEG file	jpeg	gd.c
PNG file	png	gd.c
Computer Graphics Metafile	cgm	cgm.c

Driver Functions

A dispatch table is used to direct function calls to whatever driver is chosen at run-time. Below are listed the names of each entry in the PLDispatchTable dispatch table struct defined in `plcore.h`. The entries specific to each device (defined in `drivers/*.c`) are typically named similarly but with "pl_" replaced by a string specific for that device (the logical order must be preserved, however). The dispatch table entries are :

`pl_MenuStr`: Pointer to string that is printed in device menu.

`pl_DevName`: A short device name for device selection by name.

`pl_type`: 0 for file-oriented device, 1 for interactive (the null driver uses -1 here).

`pl_init`: Initialize device. This routine may also prompt the user for certain device parameters or open a graphics file (see Notes). Called only once to set things up. Certain options such as familying and resolution (dots/mm) should be set up before calling this routine (note: some drivers ignore these).

`pl_line`: Draws a line between two points.

`pl_polyline`: Draws a polyline (no broken segments).

`pl_eop`: Finishes out current page (see Notes).

`pl_bop`: Set up for plotting on a new page. May also open a new a new graphics file (see Notes).

`pl_tidy`: Tidy up. May close graphics file (see Notes).

`pl_state`: Handle change in PLStream state (color, pen width, fill attribute, etc).

`pl_esc`: Escape function for driver-specific commands.

Notes: Most devices allow multi-page plots to be stored in a single graphics file, in which case the graphics file should be opened in the pl_init() routine, closed in pl_tidy(), and page advances done by calling pl_eop and pl_bop() in sequence. If multi-page plots need to be stored in different files then pl_bop() should open the file and pl_eop() should close it. Do NOT open files in both pl_init() and pl_bop() or close files in both pl_eop() and pl_tidy(). It is recommended that when adding new functions to only a certain driver, the escape function be used. Otherwise it is necessary to add a null routine to all the other drivers to handle the new function.

PLplot Metafiles and Plrender

The PLplot metafile is a way to store and transport your graphical data for rendering at a later time or on a different system. A PLplot metafile is in binary format in order to speed access and keep storage costs reasonable. All data is stored in device-independent format (written as a stream of bytes); the resulting file is about as portable as a tektronix vector graphics file and only slightly larger.

Each PLplot metafile begins with a header string that identifies it as such, as well as the version number of the format since this may change in time. The utility for rendering the metafile, `plrender`, verifies that the input file is indeed a valid PLplot metafile, and that it "understands" the format the metafile is written in. `plrender` is part of the PLplot package and should be built at the time of building PLplot, and then put into your search path. It is capable of high speed rendering of the graphics file, especially if the output device can accept commands at a high rate (e.g. X windows).

The commands as written by the metafile driver at present are as follows:

```
INITIALIZE

CLOSE

SWITCH_TO_TEXT

SWITCH_TO_GRAPH

CLEAR

PAGE

NEW_COLOR

NEW_WIDTH

LINE

LINETO

ESCAPE

ADVANCE
```

Each command is written as a single byte, possibly followed by additional data bytes. The `NEW_COLOR` and `NEW_WIDTH` commands each write 2 data bytes, the `LINETO` command writes 4 data bytes, and the `LINE` command writes 8 data bytes. The most common instruction in the typical metafile will be the `LINETO` command, which draws a continuation of the previous line to the given point. This data encoding is not quite as efficient as the tektronix format, which uses 4 bytes instead of 5 here (1 command + 4 data), however the PLplot encoding is far simpler to implement and more robust. The `ESCAPE` function writes a second command character (opcode) followed by an arbitrary number of data bytes depending on the value of the opcode. Note that any data written must be in device independent form to maintain the transportability of the metafile so floating point numbers are not allowed.

The short usage message for `plrender` is printed if one inputs insufficient or invalid arguments, and is as follows:

```
% plrender

No filename specified.

Usage:
        plrender [options] [files]

plrender options:
```

```
    [-v] [-i name] [-b number] [-e number] [-p page]

PLplot options:
    [-h] [-v] [-verbose] [-debug] [-dev name] [-o name] [-display name]
    [-px number] [-py number] [-geometry geom] [-wplt xl,yl,xr,yr]
    [-mar margin] [-a aspect] [-jx justx] [-jy justy] [-ori orient]
    [-freeaspect] [-width width] [-bg color] [-ncol0 n] [-ncol1 n] [-fam]
    [-fsiz size] [-fbeg number] [-finc number] [-fflen length] [-nopixmap]
    [-db] [-np] [-server_name name] [-server_host name] [-server_port name]
    [-user name]


Type plrender -h for a full description.
```

The longer usage message goes into more detail, and is as follows:

```
% plrender -h

Usage:
        plrender [options] [files]

plrender options:
    -v                  Print out the plrender version number
    -i name             Input filename
    -b number           Beginning page number
    -e number           End page number
    -p page             Plot given page only

If the "-i" flag is omitted, unrecognized input will assumed to be filename
parameters.  Specifying "-" for the input or output filename means use stdin
or stdout, respectively.  See the manual for more detail.

PLplot options:
    -h                  Print out this message
    -v                  Print out the PLplot library version number
    -verbose            Be more verbose than usual
    -debug              Print debugging info (implies -verbose)
    -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)
    -freeaspect         Do not preserve aspect ratio on orientation swaps
    -width width        Sets pen width (1 <= width <= 10)
    -bg color           Background color (0=black, FFFFFF=white)
    -ncol0 n            Number of colors to allocate in cmap 0 (upper bound)
    -ncol1 n            Number of colors to allocate in cmap 1 (upper bound)
    -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.
```

The options are generally self explanatory (family files are explained in the Section called *Family File Output*). Most of these options have default values, and for those that don't `plrender` will prompt the user. The `-px` and `-py` options are not so useful at present, because everything is scaled down by the specified factor --- resulting in labels that are too small (future versions of `plrender` might allow changing the label size as well).

Additional options may be added in future releases.

Family File Output

When sending PLplot to a file, the user has the option of generating a "family" of output files for most output file drivers. This can be valuable when generating a large amount of output, so as to not strain network or printer facilities by processing extremely large single files. Each family member file can be treated as a completely independent file. In addition, `plrender` has the ability to process a set of family member files as a single logical file.

To create a family file, one must simply call `plsfam` with the familying flag `fam` set to 1, and the desired maximum member size (in bytes) in `bmax`. `plsfam` also allows you to set the current family file number. If the current output driver does not support familying, there will be no effect. This call must be made *before* calling `plstar` or `plstart`.

If familying is enabled, the name given for the output file (on the command line, in response to the `plstar` prompt, as a `plstart` argument, or as the result of a call to `plsfnam`) becomes the stem name for the family. Thus, if you request a plmeta output file with name `test.plm`, the files actually created will be `test.plm.1`, `test.plm.2`, and so on. A new file is automatically started once the byte limit for the current file is passed, but not until the next page break. One may insure a new file at every page break by making the byte limit small enough. Alternatively, if the byte limit is large you can still insure a new file is automatically started after a page break if you precede the call to `pleop` with a call to `plfamadv`.

The `plgfam` routine can be used from within the user program to find out more about the graphics file being written. In particular, by periodically checking the number of the member file currently being written to, one can detect when a new member file is started. This information might be used in various ways; for example you could spawn a process to automatically plrender each metafile after it is closed (perhaps during a long simulation run) and send it off to be printed.

`plrender` has several options for dealing with family files. It can process a single member file (`plrender test.plm.1`) or the entire family if given only the stem name (`plrender test.plm`) It can also create family files on output, rendering to any device that supports familying, including another metafile if desired. The size of member files in this case is input through the argument list, and defaults to 1MB if unspecified (this may be changed during the PLplot installation, however). `plrender` can also create a single output file from a familied input metafile.

Interactive Output Devices

Here we shall discuss briefly some of the more common interactive output devices.

Many popular terminals or terminal emulators at present have a facility for switching between text and graphics "screens". This includes the xterm emulator under X-windows, vt100's with Retrographics, and numerous emulators for microcomputers which have a dual vt100/tek4010 emulation capability. On these devices, it is possible to switch between the text and graphics screens by surrounding your PLplot calls by calls to `plgra` and `pltext`. This will allow your diagnostic and informational code output to not interfere with your graphical output.

At present, only the xterm driver supports switching between text and graphics screens. The escape sequences as sent by the xterm driver are fairly standard, however, and have worked correctly on most other popular vt100/tek4010 emulators we've tried.

When using the xterm driver, hitting a RETURN will advance and clear the page. If indeed running from an xterm, you may resize, move, cover and uncover the window. The behavior of the X-window driver is quite different, however. First, it is much faster, as there is no tty-like handshaking going on. Second, a mouse click is used to advance and clear the page, rather than a RETURN.

On a tektronix 4014 compatible device, you may preview tektronix output files via the `pltek` utility. `pltek` will let you step through the file interactively, skipping backward or forward if desired. The help message for `pltek` is as follows:

```
% pltek
Usage: pltek filename
At the prompt, the following replies are recognized:
   h,?    Give this help message.
    q     Quit program.
   <n>    Go to the specified page number.
   -<n>   Go back <n> pages.
   +<n>   Go forward <n> pages.
 <Return> Go to the next page.
```

The output device is switched to text mode before the prompt is given, which causes the prompt to go to the vt102 window under xterm and most vt100/tek4010 emulators.

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> tk         Tcl/TK Window
 < 3> xterm      Xterm Window
 < 4> tekt       Tektronix Terminal (4010)
 < 5> tek4107t   Tektronix Terminal (4105/4107)
 < 6> mskermit   MS-Kermit emulator
 < 7> versaterm  Versaterm vt100/tek emulator
 < 8> vlt        VLT vt100/tek emulator
 < 9> plmeta     PLPLOT Native Meta-File
 <10> tekf       Tektronix File (4010)
```

```
 <11> tek4107f   Tektronix File (4105/4107)
 <12> ps         PostScript File (monochrome)
 <13> psc        PostScript File (color)
 <14> xfig       Xfig file
 <15> ljiip      LaserJet IIp/deskjet compressed graphics
 <16> ljii       LaserJet II Bitmap File (150 dpi)
 <17> 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`.

The `-dev` *`device`* command line argument, if the program's command line arguments are being passed to the PLplot function `plParseInternalOpts`.

Additional startup routines `plstar` and `plstart` 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`. 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 the Section called *Defining the Viewport* 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` 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`. This flushes any internal buffers and frees any memory that may have been allocated, for all open output streams. You may call `plend1` 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.

Adding FreeType Library Support to Bitmap Drivers

Any bitmap driver in the PLplot family should be able to use fonts (TrueType and others) that are rendered by the FreeType library just as long as the device supports setting an individual pixel. Note that drivers interact with FreeType using the support routines `plD_FreeType_init`, `plD_render_freetype_text`, `plD_FreeType_Destroy`, and `pl_set_extended_cmap0` that are coded in `plfreetype.c`.

The use of these support routines is exemplified by the `gd.c` driver. Here we make some notes to accompany this driver which should make it easier to migrate other drivers to use the FreeType library. Every code fragment we mention below should be surrounded with a `#ifdef HAVE_FREETYPE...#endif` to quarantine these fragments for systems without the FreeType library.

Write a call back function to plot a single pixel

First, write a call back function, of type `plD_pixel_fp`, which specifies how a single pixel is set in the current colour. This can be of type static void. For example, in the `gd.c` driver it looks like this:

```
void plD_pixel_gd (PLStream *pls, short x, short y)
{
png_Dev *dev=(png_Dev *)pls->dev;

   gdImageSetPixel(dev->im_out, x, y,dev->colour);
}
```

Initialise FreeType

Next, we have to initialise the FreeType library. For the `gd.c` driver this is done via two separate functions due to the order that dependent information is initialised in the driver.

The level 1 initialisation of FreeType does two things: 1) calls `plD_FreeType_init(pls)`, which in turn allocates memory to the pls- FT structure; and 2) stores the location of the call back routine.

```
void init_freetype_lv1 (PLStream *pls)
{
```

```
FT_Data *FT;

plD_FreeType_init(pls);

FT=(FT_Data *)pls->FT;
FT->pixel= (plD_pixel_fp)plD_pixel_gd;

}
```

This initialisation routine is called at the end of `plD_init_png_Dev(PLStream *pls)` in the `gd.c` driver:

```
if (freetype)
   {
    pls->dev_text = 1; /* want to draw text */
    init_freetype_lv1(pls);
    FT=(FT_Data *)pls->FT;
    FT->smooth_text=smooth_text;
   }
```

`"freetype"` is a local variable which is parsed through `plParseDrvOpts` to determine if the user wanted FreeType text. In that case `pls->dev_text` is set to 1 to indicate the driver will be rendering it's own text. After that, we always use `pls->dev_text` to work out if we want FreeType or not.

Similarly, `"smooth_text"` is a local variable passed through `plParseDrvOpts` to find out if the user wants smoothing. Since there is nothing in PLStream to track smoothing, we have to set the FT- smooth text flag as well at this time.

The level 2 initialisation function initialises everything else required for using the FreeType library but has to be called after the screen resolution and dpi have been set. Therefore, it is called at the end of `plD_init_png()`, where it looks like:

```
if (pls->dev_text)
   {
    init_freetype_lv2(pls);
   }
```

The actual function looks like this:

```
static void init_freetype_lv2 (PLStream *pls)
{
png_Dev *dev=(png_Dev *)pls->dev;
FT_Data *FT=(FT_Data *)pls->FT;

FT->scale=dev->scale;
FT->ymax=dev->pngy;
FT->invert_y=1;

if (FT->smooth_text==1)
   {
    FT->ncol0_org=pls->ncol0;                                  /* save a copy of the original size of ncol0 */
    FT->ncol0_xtra=NCOLOURS-(pls->ncol1+pls->ncol0);           /* work out how many free slots we have */
    FT->ncol0_width=FT->ncol0_xtra/(pls->ncol0-1);             /* find out how many different shades of anti-a
    if (FT->ncol0_width>64) FT->ncol0_width=64;                /* set a maximum number of shades */
    plscmap0n(FT->ncol0_org+(FT->ncol0_width*pls->ncol0));     /* redefine the size of cmap0 */
/* the level manipulations are to turn off the plP_state(PLSTATE_CMAP0)
 * call in plscmap0 which (a) leads to segfaults since the GD image is
```

```
 * not defined at this point and (b) would be inefficient in any case since
 * setcmap is always called later (see plD_bop_png) to update the driver
 * color palette to be consistent with cmap0. */
   {
      PLINT level_save;
      level_save = pls->level;
      pls->level = 0;
      pl_set_extended_cmap0(pls, FT->ncol0_width, FT->ncol0_org); /* call the function to add the extra cmap0
      pls->level = level_save;
   }
  }

}
```

FT- scale is a scaling factor to convert coordinates. This is used by the `gd.c` and some other drivers to scale back a larger virtual page and this eliminate the hidden line removal bug . Set it to 1 if your device driver doesn't use any scaling.

Some coordinate systems have zero on the bottom, others have zero on the top. FreeType does it one way, and most everything else does it the other. To make sure everything is working ok, we have to flip the coordinates, and to do this we need to know how big in the Y dimension the page is, and whether we have to invert the page or leave it alone.

FT- ymax specifies the size of the page

FT- invert y=1 tells us to invert the y-coordinates, FT- invert y=0 will not invert the coordinates.

We also do some computational gymnastics to expand cmap0 if the user wants anti-aliased text. Basically, you have to work out how many spare colours there are in the driver after cmap0 and cmap1 are done, then set a few variables in FT to let the render know how many colours it's going to have at its disposal, and call plscmap0n to resize cmap0. The call to `pl_set_extended_cmap0` does the remaining part of the work. Note it essential to protect that call by the `pls->level` manipulations for the reasons stated.

Add Function Prototypes

Next, to the top of the drivers' source file add the prototype definitions for the functions just written.

```
static void plD_pixel_gd (PLStream *pls, short x, short y);
static void init_freetype_lv1 (PLStream *pls);
static void init_freetype_lv2 (PLStream *pls);
```

Add Closing functions

Finally, add a `plD_FreeType_Destroy(pls)` entry to the device tidy function; this command deallocates memory allocated to the FT entry in the stream, closes the FreeType library and any open fonts. It is also a good idea to reset CMAP0 back to it's original size here if anti-aliasing was done. For example, in the `gd.c` driver, it looks like this:

```
void plD_tidy_png(PLStream *pls)
{
   fclose(pls->OutFile);
```

```
#ifdef HAVE_FREETYPE
   FT_Data *FT=(FT_Data *)pls->FT;
   plscmap0n(FT->ncol0_org);

   plD_FreeType_Destroy(pls);
#endif

   free_mem(pls->dev);
}
```

View Surfaces, (Sub-)Pages, Viewports and Windows

There is a whole hierarchy of coordinate systems associated with any PLplot graph. At the lowest level a device provides a view surface (coordinates in mm's) which can be a terminal screen or a sheet of paper in the output device. plinit or plstar (or plstart) makes that device view surface accessible as a page or divided up into sub-pages (see plssub) which are accessed with pladv. Before a graph can be drawn for a subpage, the program must call appropriate routines in PLplot to define the viewport for the subpage and a window for the viewport. A viewport is a rectangular region of the *subpage* which is specified in normalized subpage coordinates or millimetres. A window is a rectangular region of world-coordinate space which is mapped directly to its viewport. (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.)

Although the usual choice is to have one viewport per subpage, and one window per viewport, each subpage can have more than one (possibly overlapping) viewport defined, and each viewport can have more than one window (more than one set of world coordinates) defined.

Defining the Viewport

After defining the view surface and subpage with the appropriate call to plinit or plstar (or plstart) and a call to pladv it is necessary to define the portion of this subpage 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 using plinit or by setting `nx = 1` and `ny = 1` in plstar or plstart), 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, plsvpa, plvasp, and plvpas which are called like this:

```
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 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 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 the Section called *Setting up a Standard Window*).

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.

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 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 $0 \leq I_C \leq 10.0$ mA and $0 \leq V_{CE} \leq 12.0$ 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.

Annotating the Viewport

The routine `plbox` 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` 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 the Section called *Escape Sequences in Text*) 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`, `plschr`, `plsxax`, `plsyax`, and `plszax`. 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.

For example, let's suppose that we have set `digmax = 5` via `plsyax`, and for our plot a label is generated at `y = 0.0000478`. In this case the actual field width is longer than `digmax`, so PLplot switches to floating point. In this representation, the label is printed as simply 4.78 with the 10^{-5} exponent placed separately.

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 specification (via `digmax`). However, it will always be true that `digfix` $\leq$ `digmax`. The PLplot defaults are set up such that good results are usually obtained without user intervention.

Finally, after the call to `plbox`, the user may call routines `plgxax`, `plgyax`, or `plgzax` 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.

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);
```

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 `plcol0`. 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`. This option is not supported by all devices.

Line style is set using the routine `plstyl` or `pllsty`. 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.

Setting the Area Fill Pattern

The routine `plpat` 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` 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.

Setting Color

Normally, color is used for all drivers and devices that support it within PLplot subject to the condition that the user has the option of globally turning off the color (and subsequently turning it on again if so desired) using `plscolor`.

The PLplot color model utilizes two color maps which can be used interchangeably. However, color map0 (discussed in the Section called *Color Map0*) has discrete colors with no particular order and is most suited to coloring the background, axes, lines, and labels, and color map1 (discussed in the Section called *Color Map1*) has continuously changing colors and is most suited to plots (see the Section called *Contour and Shade Plots*) in which data values are represented by colors.

Color Map0

Color map0 is most suited to coloring the background, axes, lines, and labels. Generally, the default color map0 palette of 16 colors is used. (`examples/c/x02c.c` illustrates these colors.) The default background color is taken from the index 0 color which is black by default. The default foreground color is red.

There are a number of options for changing the default red on black colors. The user may set the index 0 background color using the command-line `bg` parameter or by calling `plscolbg` (or `plscol0` with a 0 index) *before* `plinit`. During the course of the plot, the user can change the foreground color as often as desired using `plcol0` to select the index of the desired color.

For more advanced use it is possible to define an arbitrary map0 palette of colors. The user may set the number of colors in the map0 palette using the command-line `ncol0` parameter or by calling `plscmap0n`. `plscol0` sets the RGB value of the given index which must be less than the maximum number of colors (which is set by default, by command line, by `plscmap0n`, or even by `plscmap0`). Alternatively, `plscmap0` sets up the entire map0 color palette. For all these ways of defining the map0 palette any number of colors are allowed in any order, but it is not guaranteed that the individual drivers will actually be able to use more than 16 colors.

Color Map1

Color map1 is most suited to plots (see the Section called *Contour and Shade Plots*) in which data values are represented by colors. The data are scaled to the input map1 range of floating point numbers between

0. and 1. which in turn are mapped (using `plcol1`) to colors using a default or user-specified map1 color transformation. Thus, there are calls to `plcol1` from within the code for `plshade` (see `src/plshade.c`) and `plsurf3d` (see `src/plot3d.c`) to give a continuous range of color corresponding to the data being plotted. In addition `plcol1` can be used to specify the foreground color using the map1 continuous color palette (see the commented out section of `examples/c/x12c.c` which gives an example of this for a histogram), but normally `plcol0` is a better tool for this job (see the Section called *Color Map0*) since discrete colors often give a better-looking result.

For more advanced use it is possible to define an arbitrary map1 palette of colors. The user may set the number of colors in this palette using the command-line `ncol1` parameter or by calling `plscmap1n`. Furthermore, `plscmap1l` can be used to set the map1 color palette using linear interpolation between control points specified in either RGB or HLS space.

There is a one-to-one correspondence between RGB and HLS color spaces. RGB space is characterized by three 8-bit unsigned integers corresponding to the intensity of the red, green, and blue colors. Thus, in hexidecimal notation with the 3 bytes concatanated together the RGB values of FF0000, FFFF00, 00FF00, 00FFFF, 0000FF, FF00FF, 000000, and FFFFFF correspond to red, yellow, green, cyan, blue, magenta, black, and white.

HLS (hue, lightness, and saturation) space is often conceptually easier to use than RGB space. One useful way to visualize HLS space is as a volume made up by two cones with their bases joined at the "equator". A given RGB point corresponds to HLS point somewhere on or inside the double cones, and vice versa. The hue corresponds to the "longitude" of the point with 0, 60, 120, 180, 240, and 300 degrees corresponding to red, yellow, green, cyan, blue, and magenta. The lightness corresponds to the distance along the axis of the figure of a perpendicular dropped from the HLS point to the axis. This values ranges from 0 at the "south pole" to 1 at the "north pole". The saturation corresponds to the distance of the HLS point from the axis with the on-axis value being 0 and the surface value being 1. Full saturation corresponds to full color while reducing the saturation (moving toward the axis of the HLS figure) mixes more gray into the color until at zero saturation on the axis of the figure you have only shades of gray with the variation of lightness along the axis corresponding to a gray scale.

Here are some C-code fragments which use `plscmap1l` to set the map1 color palette. This first example illustrates how to set up a gray-scale pallette using linear interpolation in RGB space.

```
i[0] = 0.;
i[1] = 1.;
/* RGB are rescaled to the range from 0 to 1. for input to plscmap1l.*/
r[0] = 0.;
r[1] = 1.;
g[0] = 0.;
g[1] = 1.;
b[0] = 0.;
b[1] = 1.;
plscmap1l(1, 2, i, r, g, b, NULL);
```

This second example illustrates doing the same thing in HLS space.

```
i[0] = 0.;
i[1] = 1.;
/* Hue does not matter for zero saturation.*/
h[0] = 0.;
h[1] = 0.;
/* Lightness varies through its full range.*/
l[0] = 0.;
l[1] = 1.;
/* Saturation is zero for a gray scale.*/
```

```
    s[0] = 0.;
    s[1] = 0.;
    /* Note the first argument which specifies HLS space.*/
    plscmap1l(0, 2, i, h, l, s, NULL);
```

This final example using `plscmap1l` illustrates how the default map1 color pallette is set with just 4 control points (taken from `src/plctrl.c`).

```
/*--------------------------------------------------------------------------*\
 * plcmap1_def()
 *
 * Initializes color map 1.
 *
 * The default initialization uses 4 control points in HLS space, the two
 * inner ones being very close to one of the vertices of the HLS double
 * cone.  The vertex used (black or white) is chosen to be the closer to
 * the background color.  If you don't like these settings you can always
 * initialize it yourself.
\*--------------------------------------------------------------------------*/

static void
plcmap1_def(void)
{
    PLFLT i[4], h[4], l[4], s[4], vertex = 0.;

/* Positions of control points */

    i[0] = 0;  /* left boundary */
    i[1] = 0.45; /* just before center */
    i[2] = 0.55; /* just after center */
    i[3] = 1;  /* right boundary */

/* For center control points, pick black or white, whichever is closer to bg */
/* Be carefult to pick just short of top or bottom else hue info is lost */

    if (plsc->cmap0 != NULL)
 vertex = ((float) plsc->cmap0[0].r +
    (float) plsc->cmap0[0].g +
    (float) plsc->cmap0[0].b) / 3. / 255.;

    if (vertex < 0.5)
 vertex = 0.01;
    else
 vertex = 0.99;

/* Set hue */

    h[0] = 260;  /* low: blue-violet */
    h[1] = 260;  /* only change as we go over vertex */
    h[2] = 0;  /* high: red */
    h[3] = 0;  /* keep fixed */

/* Set lightness */

    l[0] = 0.5;  /* low */
    l[1] = vertex; /* bg */
    l[2] = vertex; /* bg */
```

```
    l[3] = 0.5;  /* high */

/* Set saturation -- keep at maximum */

    s[0] = 1;
    s[1] = 1;
    s[2] = 1;
    s[3] = 1;

    c_plscmap1l(0, 4, i, h, l, s, NULL);
}
```

Finally, plscmap1 is an additional method of setting the map1 color palette directly using RGB space. No interpolation is used with plscmap1 so it is the programmer's responsibility to make sure that the colors vary smoothly. Here is an example of the method taken from examples/c/x08c.c which sets (yet again) the gray-scale color pallette.

```
for (i=0;i<n_col;i++)
    rr[i] = gg[i] = bb[i] = i*256/n_col;
plscmap1(rr,gg,bb,n_col);
```

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. 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 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 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` 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` and `plsmin`.

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 3-3 shows how these letters map into Greek characters.

Table 3-3. Roman Characters Corresponding to Greek Characters

Roman	A	B	G	D	E	Z	Y	H	I	K	L	M
Greek	*A*	*B*	`	´	*E*	*Z*	*E*	ˆ	*I*	*K*	˜	*M*
Roman	N	C	O	P	R	S	T	U	F	X	Q	W
Greek	*N*	¨	*O*	″	*R*	°	*T*	˘	˘	*C*	¯	·
Roman	a	b	g	d	e	z	y	h	i	k	l	m

Greek	α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ
Roman	n	c	o	p	r	s	t	u	f	x	q	w
Greek	ν	ξ	o	π	ρ	σ	τ	υ	ϕ	χ	ψ	ω

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` or `plenv` is used in the usual way to establish the size of the two-dimensional window. The routine `plw3d` 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`.

For example, if the three-dimensional world-coordinate ranges are -10.0 $\leq$ `x` $\leq$ 10.0, -3.0 $\leq$ `y` $\leq$ +7.0, and 0.0 $\leq$ `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 row 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` 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`.

Labeling 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` thus combines the functions of box drawing and labeling.

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 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.

Contour Plots from C

`plcont` is the routine callable from C for plotting contours. This routine has the form:

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

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.

The path of each contour is initially computed in terms of the values of the array indices 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 the Section called *Defining the Window*), it is necessary to convert from these array indices into world coordinates. This is done by passing a pointer `pltr` to a user-defined transformation function to `plcont`. For C use of `plcont` (and `plshade`, see next subsection) we have included directly in the PLplot library the following transformation routines: `pltr0` (identity transformation or you can enter a NULL argument to get the same effect); `pltr1` (linear interpolation in singly dimensioned coordinate arrays); and `pltr2` (linear interpolation in doubly dimensioned coordinate arrays). Examples of the use of these transformation routines are given in `examples/c/x09c.c`, `examples/c/x14c.c`, and `examples/c/x16c.c`. These same three examples also demonstrate a user-defined transformation function `mypltr` which is capable of arbitrary translation, rotation, and/or shear. By defining other transformation subroutines, it is possible to draw contours wrapped around polar grids etc.

Shade Plots from C

NEEDS DOCUMENTATION

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 `plcontfortran` from Fortran is then:

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

When called from Fortran, this routine has the same effect as when invoked from C. The interpretation of all parameters (see `plcont`) is also the same except there is no transformation function supplied as the last parameter. Instead, a 6-element array specifying coefficients to use in the transformation is supplied via the named common block `plplot` (see code). Since this approach is somewhat inflexible, the user is recommended to call either of `plcon0`, `plcon1`, or `plcon2` instead.

The three routines recommended for use from Fortran are `plcon0`, `plcon1`, and `plcon2`. 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 as well as a call to plcont.

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
```

assuming the user had already set up arrays `r` and `theta` to specify the (r, θ) values at the gridpoints in his system. For this example, it is recommended that the user add an additional cell in theta such that `xg(i, NY+1) = xg(i, 1)` and `yg(i, NY+1) = yg(i, 1)` so that the contours show the proper periodic behavior in θ (see also 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
```

Shade Plots from Fortran

NEEDS DOCUMENTATION

Chapter 4. The PLplot X Driver Family

There are several drivers for working with the MIT X Windows system. Beginning with PLplot 5.0, the following drivers are supported.

The Xwin Driver

is cool.

The Tk Driver

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

Chapter 5. The PLplot Output Driver Family

There are several drivers which produce output files to be displayed by another system. Beginning with PLplot 5.0, the following drivers are supported.

The Postscript Driver

makes .ps output, how cool.

III. Language Bindings

Chapter 6. C Language

(OLD, NEEDS DOCUMENTATION UPDATING) The argument types given in this manual (PLFLT and PLINT) are typedefs for the actual argument type. A PLINT is actually a type `long` and should not be changed. A PLFLT can be either a `float` or `double`; this choice is made when the package is installed and on a Unix system (for example) may result in a PLplot library named `libplplot.a` in single precision and `libplplotd.a` in double precision.

These and other constants used by PLplot are defined in the main header file `plplot.h`, which must be included by the user program. This file also contains all of the function prototypes, machine dependent defines, and redefinition of the C-language bindings that conflict with the Fortran names (more on this later). `plplot.h` obtains its values for PLFLT, PLINT, and PLARGS (a macro for conditionally generating prototype argument lists) from FLOAT (typedef), INT (typedef), and PROTO (macro), respectively. The latter are defined in the file `chdr.h`. The user is encouraged to use FLOAT, INT, and PROTO in his/her own code, and modify `chdr.h` according to taste. It is not actually necessary to declare variables as FLOAT and INT except when they are pointers, as automatic conversion to the right type will otherwise occur (if using a Standard C compiler; else K&R style automatic promotion will occur). The only code in `plplot.h` that directly depends on these settings is as follows:

```
#include "plplot/chdr.h"

/* change from chdr.h conventions to plplot ones */

typedef FLOAT PLFLT;
typedef INT   PLINT;
#define PLARGS(a) PROTO(a)
```

PLplot is capable of being compiled with Standard C (ANSI) mode on or off. This is toggled via the macro PLSTDC, and set automatically if STDC is defined. If PLSTDC is defined, all functions are prototyped as allowed under Standard C, and arguments passed exactly as specified in the prototype. If PLSTDC is not defined, however, function prototypes are turned off and K&R automatic argument promotion will occur, e.g. `float &rarr; double, int &rarr; long`. There is no middle ground! A PLplot library built with PLSTDC defined will not work (in general) with a program built with PLSTDC undefined, and vice versa. It is possible in principle to build a library that will work under both Standard C and K&R compilers simultaneously (i.e. by duplicating the K&R promotion with the Standard C prototype), but this seems to violate the spirit of the C standard and can be confusing. Eventually we will drop support for non-standard C compilers but for now have adopted this compromise.

In summary, PLplot will work using either a Standard or non-standard C compiler, provided that you :

- Include the PLplot main header file `plplot.h`.
- Make sure all pointer arguments are of the correct type (the compiler should warn you if you forget, so don't worry, be happy).
- Do not link a code compiled with PLSTDC defined to a PLplot library compiled with PLSTDC undefined, or vice versa.
- Use prototypes whenever possible to reduce type errors.

Note that some Standard C compilers will give warnings when converting a constant function argument to whatever is required by the prototype. These warnings can be ignored.

The one additional complicating factor concerns the use of stub routines to interface with Fortran (see the following section for more explanation). On some systems, the Fortran and C namespaces are set up to clobber each other. More reasonable (from our viewpoint) is to agree on a standard map between namespaces, such as the appending of an underscore to Fortran routine names as is common on many Unix-like systems. The only case where the shared Fortran/C namespaces do any good is when passing a pointer to a like data type, which represents only a small fraction of the cases that need to be handled (which includes constant values passed on the stack, strings, and two-dimensional arrays).

There are several ways to deal with this situation, but the least messy from a user's perspective is to redefine those PLplot C function names which conflict with the Fortran-interface stub routines. The actual function names are the same as those described in this document, but with a "c_" prepended. These macro definitions appear in the `plplot.h` header file and are otherwise harmless. Therefore you can (and should) forget that most of the names are being redefined to avoid the conflict and simply adhere to the bindings as described in this manual. Codes written under old versions of PLplot (previous to 5.0) will require a recompile, however.

For more information on calling PLplot from C, please see the example C programs (`x01c.c` through `x19c.c`) distributed with PLplot.

Chapter 7. Fortran Language

As discussed in the preceding section, PLplot's integer representation is a PLINT and its floating point representation is a PLFLT. To the Fortran user, this most commonly translates to a type `integer` and type `real`, respectively. This is somewhat system dependent (and up to the installer of the package) so you should check the release notes to be sure, or just try it and see what happens.

Because the PLplot kernel is written in C, standard C syntax is used in the description of each PLplot function. Thus to understand this manual it is helpful to know a little about C, but fortunately the translation is very easy and can be summarized here. As an example, the routine `plline` call from C would look like:

```
plline(n,x,y);
```

while from Fortran it would look like:

```
call plline(n,x,y)
```

typically with `n` declared as type `integer` and `x`, `y` declared as type `real` (arrays in this case). Each C language type used in the text translates roughly as follows:

PLFLT	real
PLINT	integer
char *	character
PLFLT *	real or real array
PLFLT **	real array
`"string"`	`'string'`
array[0]	array(1)

In C there are two ways to pass a variable --- by value (the default) or by reference (pointer), whereas only the latter is used by Fortran. Therefore when you see references in the text to *either* an ordinary argument or a pointer argument (e.g. `*data`), you simply use an ordinary Fortran variable or array name.

The PLplot library comes with a set of Fortran interface routines that allow the exact same call syntax (usually) regardless of whether calling from C or Fortran. In some cases, this means the subroutine name *exceeds 8 characters in length*. Nearly every Fortran compiler available today allows subroutine names longer than 8 characters, so this should not be a problem (although if it ever is, in principle a truncated name could be defined for that platform).

These "stub" routines handle transforming the data from the normal Fortran representation to that typically used in C. This includes:

Variables passed by value instead of by reference.

Fortran passes all subroutine arguments by reference, i.e., a pointer to the argument value is pushed on the stack. In C all values, except for arrays (including char arrays), are passed by value, i.e., the argument value itself is pushed on the stack. The stub routine converts the Fortran call by reference to a call by value. As an example, here is how the plpoin stub routine works. In your Fortran program you might have a call to plpoin that looks something like

```
call plpoin(6,x,y,9)
```

where x and y are arrays with 6 elements and you want to plot symbol 9. As strange as it seems (at least to C programmers) the constants 6 and 9 are passed by reference. This will actually call the following C stub routine (included in entirety)

```
#include "plplot/plstubs.h"
```

```
void
PLPOIN(n, x, y, code)
PLINT *n, *code;
PLFLT *x, *y;
{
    c_plpoin(*n, x, y, *code);
}
```

All this stub routine does is convert the number of points (`*n` and the symbol `*code` to call by value (i.e. pushes their value on the stack) and then calls the C plpoin library routine.

Get mapping between Fortran and C namespace right (system dependent).

The external symbols (i.e. function and subroutine names) as you see them in your program often appear differently to the linker. For example, the Fortran routine names may be converted to uppercase or lowercase, and/or have an underscore appended or prepended. This translation is handled entirely via redefinition of the stub routine names, which are macros. There are several options for compiling PLplot that simplify getting the name translation right (NEEDS DOCUMENTATION IF THESE STILL EXIST). In any case, once the name translation is established during installation, name translation is completely transparent to the user.

Translation of character string format from Fortran to C.

Fortran character strings are passed differently than other quantities, in that a string descriptor is pushed on the stack along with the string address. C doesn't want the descriptor, it wants a NULL terminated string. For routines that handle strings two stub routines are necessary, one written in Fortran and one written in C. Your Fortran program calls the Fortran stub routine first. This stub converts the character string to a null terminated integer array and then calls the C stub routine. The C stub routine converts the integer array (type `long`) to the usual C string representation (which may be different, depending on whether your machine uses a big endian or little endian byte ordering; in any case the way it is done in PLplot is portable). See the `plmtex` stubs for an example of this.

Note that the portion of a Fortran character string that exceeds 299 characters will not be plotted by the text routines (`plmtex` and `plptex`).

Multidimensional array arguments are changed from row-dominant to column-dominant ordering through use of a temporary array.

In Fortran, arrays are always stored so that the first index increases most rapidly as one steps through memory. This is called "row-dominant" storage. In C, on the other hand, the first index increases *least* rapidly, i.e. "column-dominant" ordering. Thus, two dimensional arrays (e.g. as passed to the contour or surface plotting routines) passed into PLplot must be transposed in order to get the proper two-dimensional relationship to the world coordinates. This is handled in the C stub routines by dynamic memory allocation of a temporary array. This is then set equal to the transpose of the passed in array and passed to the appropriate PLplot routine. The overhead associated with this is normally not important but could be a factor if you are using very large 2d arrays.

This all seems a little messy, but is very user friendly. Fortran and C programmers can use the same basic interface to the library, which is a powerful plus for this method. The fact that stub routines are being used is completely transparent to the Fortran programmer.

For more information on calling PLplot from Fortran, please see the example Fortran programs (`x01f.f` through `x16f.f`) distributed with PLplot.

Chapter 8. A C++ Interface for PLplot

PLplot has long had C and Fortran bindings, presenting a fairly conventional API to the applications programmer. Recently (1994 onwards) PLplot has been growing interfaces (language bindings) to a variety of other languages. In this chapter we discuss the PLplot C++ support provided in the PLplot distribution. Of course many other approaches are possible, perhaps even in use by PLplot users around the world. The purpose of this chapter then is to explain the rationale and intended usage for the bundled C++ language support.

Motivation for the C++ Interface

PLplot has a fairly complex C API. There are lots of functions, and several facilities have multiple entry points with similar names but different argument lists. (Think contouring, shading). Often these differing argument lists are to accommodate a variety of data storage paradigms, one of which you are expected to be using!

Especially in the case of the 2-d API's for contouring and shading, sophisticated C++ users may feel a special sense of exasperation with the data layout prescriptions, since they are extremely primitive, pointer rich, and prone to a wide class of memory leaks and other sorts of programming errors. Many C++ users know good and well that better ways exist (templated matrix classes, etc), but historically have not been able to use these more sophisticated techniques if the contained data ever needed to get plotted.

Besides the 2-d API functions, there is also the multiple output stream capability of PLplot. Anyone who knows C++ well, and who has used multiple output streams in PLplot, has probably noticed striking similarities between the PLplot `PLStream` pointer and the C++ `this` pointer. Although multiple output streams have not been widely used in PLplot applications in the past, the availability of the plframe Tk widget, and the extended wish concept, is making it much more attractive to use multiple output streams.

Unfortunately, if you do write a Tk extended wish application, and endow your interface with multiple plframes, the event driven character of X applications makes it difficult to ensure that PLplot output shows up in the right plframe window. If a plot is generated to one plframe, the PLplot `PLStream` pointer is directed to that stream. If a user then pushes a Tk button which should generate a plot to a different plframe, the plot goes to the old plframe instead! Schemes for controlling this can be imagined, but the logic can be complex, especially in the face of the ability to /also/ make plots to the same plframe from either Tcl or C++.

Beyond this, the C API is downright ugly for a significant number of the functions, particularly those which return values by accepting pointers to variables in their argument lists, and then changing them in that way. Sophisticated C++ users generally take considerable pride in banishing the offensive bare pointer from their code, and consider it disgusting to have to insert &'s just in order to make a call to an API function.

In order to address these issues (and more), I have begun constructing a C++ interface to PLplot. The purpose of this missive is to describe its architecture and usage.

Design of the PLplot C++ Interface

Stream/Object Identity

A C++ class named `plstream` has been introduced. It's central purpose is provide a specific, object based encapsulation of the concept of a PLplot output stream. Any output produced using a `plstream` object,

will go to the PLplot output stream associated with that object, regardless of what stream may have been active before.

In order to write a multiple output stream PLplot application, a C++ program can declare `plstream` objects, and invoke drawing methods on those objects, without regard to ordering considerations or other coherency considerations. Although this has obvious simplification benefit even for simple programs, the full benefit is most easily appreciated in the context of Tk extended wish applications in which a `plstream` can be associated with each plframe.

Namespace Management

The PLplot C API is composed of a set of drawing functions, all prefixed with pl , in an effort to prevent namespace collision. However, the prefix pl is gratuitous, and in particular is unnecessary in a C++ context. The `plstream` class mirrors most of the PLplot C API, but does so by dropping the pl prefix. The `plstream` class thus serves to collect the PLplot drawing functions into a scope in which collisions with other similarly named functions is not a concern. So, where a C programmer might write:

```
       plsstrm( 1 );
plenv( ... );
plline( ... );
```

The C++ programmer can write:

```
plstream p( ... );
p.env( ... );
p.line( ... );
```

Is that an important benefit? The utility varies with the number of output streams in use in the program.

plmkstrm() is replaced by object declaration. plsstrm() is replaced by method invocation on the desired output stream object. plgstrm() is rendered irrelevant.

The skeptic may say, But you have to type the same number of characters! You've replaced 'pl' with 'p.', except it could be worse for a longer object name. True. BUT, in this new scheme, most plots will not be generated by invoking methods on a specific stream object, but rather by deriving from `plstream`, and invoking methods of this object. See the section on derivation below.

Abstraction of Data Layout

The `plstream` class will provide an abstract interface to the 2-d drawing functions. Instead of forcing the C++ user to organize data in one of a small set of generally braindead data layouts with poor memory management properties, potentially forcing the C++ user to not use a superior method, or to copy data computed in one layout format to another for plotting (with consequent bug production), the `plstream` 2-d plotting functions will accept an abstract layout specification. The only thing which is important to the 2-d drawing functions is that the data be indexable . They should not care about data layout.

Consequently, an abstract class, Contourable Data is provided. This class provides a pure virtual method which accepts indexes, and is to be made to produce a function value for the user's 2-d data field. It is of no concern to PLplot how the user does this. Any mapping between index and data which the user wishes to use, may be used.

This methodology allows the C++ user to compute data using whatever storage mechanism he wants. Then, by deriving a class from PLplot's Contourable Data abstract class, he can provide a mapping to his own data layout.

Note that this does /not/ mean that the C++ user's internal data layout must be derived from PLplot's Contourable Data class. Suppose for example that the user data is stored in a C++ matrix class. To

make this data contourable, the user may define a class which specializes the indexing concept of the PLplot Contourable Data class to his matrix class. For example:

```
class Matrix { ... };
class Contourable_Matrix : public Contourable_Data {
    Matrix& m;
  public:
    Contourable_Matrix( Matrix& _m ) : m(_m) {}
    PLFLT  operator()( int i, int j ) const { return m(i,j); }
};

plstream p( ... );
Matrix m;
// Code to fill m with data
Contourable_Matrix cm(m);
p.shade( cm, ... );
```

In this way the C++ user is completely freed from the tyranny of moronic data layout constraints imposed by PLplot's C or Fortran API.

Collapsing the API

Use of abstraction as in C) above will allow a single method in `plstream` to perform the services of multiple functions in the C API. In those cases where multiple functions were provided with different data layout specifications, but similar functionality, these can all be collapsed into one, through the use of the abstract interface technique described above. Moreover, function name overloading can be used to simplify the namespace for those cases where multiple functions were used to get variations on a basic capability. For example, a single name such as contour or shade can be used for multiple methods taking different argument sets, so that for example, one can make simple plots of rectangular data sets, or more complex generalized coordinate mappings.

Specializing the PLplot C++ Interface

The `plstream` class is an ideal candidate for derivation. By inheriting from `plstream`, the user can construct a new class which is automatically endowed with the ability to plot to a specific PLplot output stream in a coherent manner without having to worry about interplay with other `plstream` (or derived type) objects. Moreover, new, higher level, plotting functionality can be constructed to provide even more simplicity and ease of use than the PLplot API.

The PLplot maintainers (Geoff and Maurice) expect to introduce a class plxstream in the future which provides superior support for constructing graphics with multiple plots per page, easier specification of plot adornments, etc. This should significantly ease one aspect of PLplot usage which we regard as being clumsy at this time.

Beyond that, users may find it useful to derive from `plstream` (or later plxstream whenever it finally makes its appearance) for the purpose of making application specific output streams. For example, a C++ program will normally have a variety of objects which constitute the fundamental entities in the code. These could all be made to be atomically plotted by providing suitable methods. For example:

```
class Cat { ... };
class Dog { ... };
class Bear { ... };
class Fish { ... };

class zoostream : public plstream {
  public:
```

```
    void plot( const Cat& c ) { ... }
    void plot( const Dog& d ) { ... }
    void plot( const Bear& b ) { ... }
    void plot( const Fish& f ) { ... }
};
```

Presumably the PLplot user community can think of even more imaginative uses... :-).

Status of the C++ Interface

The class `plstream` (and the other abstraction classes in `plstream.h`) provided in PLplot 4.99j (alpha) are to be considered as works in progress. By the standards outlined above, the work has barely begun. At this time, `plstream` is mostly a one to one mirror of the C API, which is to say, it is still far from the goals of simplification and abstraction outlined above. As such, it can be expected to change radically over the course of time. (We don't quote schedules--how long have you been waiting for 5.0? :-).

In any event, we would welcome improvement submissions along the lines of those above, but we would strongly discourage people from using `plstream` if they are expecting it to be rock solid. It *will* be changing, to become more like the design goals elucidated above.

So, if you like the ideas described above, and are willing to accept the burden of upgrading your code as the class `plstream` evolves, then feel free to use it. Just don't whine when I fix some of the methods to take references instead of pointers, when I eliminate some of the redundant methods to use the collapsed form, etc.

Chapter 9. Using PLplot from Tcl

PLplot has historically had C and Fortran language bindings. PLplot version 5.0 introduces a plethora of new programming options including C++ (described earlier) and several script language bindings. The Tcl interface to PLplot (which the PLplot maintainers regard as the "primary" script language binding) is described in this chapter, with further discussion of Tcl related issues following in additional chapters. But Tcl is certainly not the only script language option. Bindings to Perl, Python, and Scheme (which is actually another compiled language, but still has some of the flavor of a VHLL) are in various stages of completion, and are described in separate chapters. Use the one that suits you best--or try them all!

Motivation for the Tcl Interface to PLplot

The recent emergence of several high quality VHLL script languages such as Tcl, Perl, Python and arguably even some Lisp variants, is having a profound effect upon the art of computer programming. Tasks which have traditionally been handled by C or Fortran, are beginning to be seen in a new light. With relatively fast processors now widely available, many programming jobs are no longer bound by execution time, but by "human time". Rapidity of initial development and continued maintenance, for a surprisingly wide class of applications, is far more important than execution time. Result: in a very short period of time, say from 1993 to 1995, script languages have exploded onto the scene, becoming essential tools for any serious programmer.

Moreover, the entire concept of "speed of execution" needs revising in the face of the gains made in computer hardware in recent years. Saying that script language processing is slower than compiled language processing may be undeniable and simultaneously irrelevant. If the script language processing is fast enough, then it is fast enough. Increasingly, computational researchers are finding that script based tools are indeed fast enough. And if their run time is fast enough, and their development and maintenance time is much much better, then why indeed should they not be used?

Even in a field with several high visibility players, Tcl has distinguished itself as a leading contender. There are many reasons for this, but perhaps the most important, at least as it relates to the PLplot user community, is that Tcl was designed to be extensible and embeddable. The whole purpose of Tcl, as it name (Tool Command Language) indicates, is to be a command language for other tools. In other words, the fact that Tcl is capable of being a standalone shell is interesting, even useful, but nonetheless incidental. The real attraction of Tcl is that it can be the shell language for *your* code. Tcl can easily be embedded into your code, endowing it immediately with a full featured, consistent and well documented script programming language, providing all the core features you need in a programming language: variables, procedures, control structures, error trapping and recovery, tracing, etc. But that is only the beginning! After that, you can easily extend Tcl by adding commands to the core language, which invoke the capabilities of your tool. It is in this sense that Tcl is a tool command language. It is a command language which you can augment to provide access to the facilities of your tool.

But Tcl is more than just an embeddable, extensible script language for personal use. Tcl is an industry, an internet phenomenon. There are currently at least two high quality books, with more on the way. There is an industry of service providers and educators. Furthermore, literally hundreds of Tcl extensions exist, and are readily available over the net. Perhaps the most notable extension, Tk, provides a fantastic interface to X Windows widget programming, permitting the construction of Motif like user interfaces, with none of the hassles of actually using Motif. Some of these extensions endow Tcl with object oriented facilities philosophically similar to C++ or other object oriented languages. Other extensions provide script level access to system services. Others provide a script interface to sockets, RPC, and other network programming protocols. The list goes on and on. Dive into the Tcl archive, and see what it has for you!

So, the answer to the question "Why do we want a Tcl interface to PLplot?" is very simple. "Because we we are using Tcl anyway, as the command language for our project, and would like to be able to do

plotting in the command language just as we do so many other things."

But there is more than just the aesthetics of integration to consider. There are also significant pragmatic considerations. If you generate your PLplot output via function calls from a compiled language, then in order to add new diagnostics to your code, or to refine or embellish existing ones, you have to edit the source, recompile, relink, and rerun the code. If many iterations are required to get the plot right, significant time can be wasted. This can be especially true in the case of C++ code making heavy use of templates, for which many C++ compilers will have program link times measured in minutes rather than seconds, even for trivial program changes.

In contrast, if the diagnostic plot is generated from Tcl, the development cycle looks more like: start the shell (command line or windowing), source a Tcl script, issue the command to generate the plot, notice a bug, edit the Tcl script, resource the script, and regenerate the plot. Notice that compiling, linking, and restarting the program, have all been dropped from the development cycle. The time savings from such a development cycle can be amazing!

Overview of the Tcl Language Binding

Each of the PLplot calls available to the C or Fortran programmer are also available from Tcl, with the same name and generally the same arguments. Thus for instance, whereas in C you can write:

```
plenv( 0., 1., 0., 1., 0, 0 );
pllab( "(x)", "(y)", "The title of the graph" );
```

you can now write in Tcl:

```
plenv 0 1 0 1 0 0
pllab "(x)" "(y)" "The title of the graph"
```

All the normal Tcl rules apply, there is nothing special about the PLplot extension commands. So, you could write the above as:

```
set xmin 0; set xmax 1; set ymin 0; set ymax 1
set just 0; set axis 0
set xlab (x)
set ylab (y)
set title "The title of the graph"
plenv $xmin $xmax $ymin $ymax $just $axis
pllab $xlab $ylab $title
```

for example. Not that there is any reason to be loquacious for its own sake, of course. The point is that you might have things like the plot bounds or axis labels stored in Tcl variables for some other reason (tied to a Tk entry widget maybe, or provided as the result of one of your application specific Tcl extension commands, etc), and just want to use standard Tcl substitution to make the PLplot calls.

Go ahead and try it! Enter `pltcl` to start up the PLplot extended Tcl shell, and type (or paste) in the commands. Or put them in a file and source it. By this point it should be clear how incredibly easy it is to use the PLplot Tcl language binding.

In order to accommodate the ubiquitous requirement for matrix oriented data in scientific applications, and in the PLplot API in particular, PLplot 5.0 includes a Tcl extension for manipulating matrices in Tcl. This Tcl Matrix Extension provides a straightforward and direct means of representing one and two dimensional matrices in Tcl. The Tcl Matrix Extension is described in detail in the next section, but we mention its existence now just so that we can show how the PLplot Tcl API works. Many of the PLplot Tcl API functions accept Tcl matrices as arguments. For instance, in C you might write:

```
float x[100], y[100];

/* code to initialize x and y */
```

```
plline( 100, x, y );
```

In Tcl you can write:

```
matrix x f 100
matrix y f 100

# code to initialize x and y

plline 100 x y
```

Some of the PLplot C function calls use pointer arguments to allow retrieval of PLplot settings. These are implemented in Tcl by changing the value of the variable whose name you provide. For example:

```
pltcl> plgxax
wrong # args: should be "plgxax digmax digits  "
pltcl> set digmax 0
0
pltcl> set digits 0
0
pltcl> plgxax digmax digits
pltcl> puts "digmax=$digmax digits=$digits"
digmax=4 digits=0
```

This example shows that each PLplot Tcl command is designed to issue an error if you invoke it incorrectly, which in this case was used to remind us of the correct arguments. We then create two Tcl variables to hold the results. Then we invoke the PLplot `plgxax` function to obtain the label formatting information for the x axis. And finally we print the results.

People familiar with Tcl culture may wonder why the `plg*` series functions don't just pack their results into the standard Tcl result string. The reason is that the user would then have to extract the desired field with either `lindex` or `regexp`, which seems messy. So instead, we designed the PLplot Tcl API to look and feel as much like the C API as could reasonably be managed.

In general then, you can assume that each C function is provided in Tcl with the same name and same arguments (and one or two dimensional arrays in C are replaced by Tcl matrices). There are only a few exceptions to this rule, generally resulting from the complexity of the argument types which are passed to some functions in the C API. Those exceptional functions are described below, all others work in the obvious way (analogous to the examples above).

See the Tcl example programs for extensive demonstrations of the usage of the PLplot Tcl API. To run the Tcl demos:

```
% pltcl
pltcl> source tcldemos.tcl
pltcl> 1
pltcl> 2
```

Alternatively, you can run `plserver` and source `tkdemos.tcl`.

In any event, the Tcl demos provide very good coverage of the Tcl API, and consequently serve as excellent examples of usage. For the most part they draw the same plots as their C counterpart. Moreover, many of them were constructed by literally inserting the C code into the Tcl source file, and performing fairly mechanical transformations on the source. This should provide encouragement to anyone used to using PLplot through one of the compiled interfaces, that they can easily and rapidly become productive with PLplot in Tcl.

The PLplot Tcl Matrix Extension

Tcl does many things well, but handling collections of numbers is not one of them. You could make lists, but for data sets of sizes relevant to scientific graphics which is the primary domain of applicability for PLplot, the extraction time is excessive and burdensome. You could use Tcl arrays, but the storage overhead is astronomical and the lookup time, while better than list manipulation, is still prohibitive.

To cope with this, a Tcl Matrix extension was created for the purpose of making it feasible to work with large collections of numbers in Tcl, in a way which is storage efficient, reasonably efficient for accesses from Tcl, and reasonably compatible with practices used in compiled code.

Using Tcl Matrices from Tcl

Much like the Tk widget creation commands, the Tcl `matrix` command considers its first argument to be the name of a new command to be created, and the rest of the arguments to be modifiers. After the name, the next argument can be `float` or `int` or contractions thereof. Next follow a variable number of size arguments which determine the size of the matrix in each of its dimensions. For example:

```
matrix x f 100
matrix y i 64 64
```

constructs two matrices. `x` is a float matrix, with one dimension and 100 elements. `y` is an integer matrix, and has 2 dimensions each of size 64.

Additionally, an initializer may be specified, with a syntax familiar from C. For example:

```
matrix x f 4 = { 1.5, 2.5, 3.5, 4.5 }
```

A Tcl matrix is a command, and as longtime Tcl users know, Tcl commands are globally accessible. The PLplot Tcl Matrix extension attempts to lessen the impact of this by registering a variable in the local scope, and tracing it for insets, and deleting the actual matrix command when the variable goes out of scope. In this way, a Tcl matrix appears to work sort of like a variable. It is, however, just an illusion, so you have to keep this in mind. In particular, you may want the matrix to outlive the scope in which it was created. For example, you may want to create a matrix, load it with data, and then pass it off to a Tk megawidget for display in a spreadsheet like form. The proc which launches the Tk megawidget will complete, but the megawidget, and the associated Tcl matrix are supposed to hang around until they are explicitly destroyed. To achieve this effect, create the Tcl matrix with the `-persist` flag. If present (can be anywhere on the line), the matrix is not automatically deleted when the scope of the current proc (method) ends. Instead, you must explicitly clean up by using either the 'delete' matrix command or renaming the matrix command name to . Now works correctly from within [incr Tcl].

As mentioned above, the result of creating a matrix is that a new command of the given name is added to the interpreter. You can then evaluate the command, providing indices as arguments, to extract the data. For example:

```
pltcl> matrix x f = {1.5, 2.5, 3.5, 4.5}
insufficient dimensions given for Matrix operator "x"
pltcl> matrix x f 4 = {1.5, 2.5, 3.5, 4.5}
pltcl> x 0
1.500000
pltcl> x 1
2.500000
pltcl> x 3
4.500000
pltcl> x *
1.500000 2.500000 3.500000 4.500000
pltcl> puts "x\[1\]=[x 1]"
x[1]=2.500000
pltcl> puts "x\[*\] = :[x *]:"
```

```
x[*] = :1.500000 2.500000 3.500000 4.500000:
pltcl> foreach v [x *] { puts $v }
1.500000
2.500000
3.500000
4.500000
pltcl> for {set i 0} {$i < 4} {incr i} {
           if {[x $i] < 3} {puts [x $i]} }
1.500000
2.500000
```

Note from the above that the output of evaluating a matrix indexing operation is suitable for use in condition processing, list processing, etc.

You can assign to matrix locations in a similar way:

```
pltcl> x 2 = 7
pltcl> puts ":[x *]:"
:1.500000 2.500000 7.000000 4.500000:
pltcl> x * = 3
pltcl> puts ":[x *]:"
```

Note that the * provides a means of obtaining an index range, and that it must be separated from the = by a space. Future versions of the Tcl Matrix extension may allow alternative ways of specifying index ranges and may assign the obvious meaning to an expression of the form:

```
x *= 3
```

However this has not been implemented yet...

In any event, the `matrix` command also supports an `info` subcommand which reports the number of elements in each dimension:

```
pltcl> x info
4
pltcl> matrix y i 8 10
pltcl> y info
8 10
```

Using Tcl Matrices from C

Normally you will create a matrix in Tcl, and then want to pass it to C in order to have the data filled in, or existing data to be used in a computation, etc. To do this, pass the name of the matrix command as an argument to your C Tcl command procedure. The C code should include `tclMatrix.h`, which has a definition for the `tclMatrix` structure. You fetch a pointer to the `tclMatrix` structure using the `Tcl_GetMatrixPtr` function.

For example, in Tcl:

```
matrix x f 100
wacky x
```

and in C:

```
int wackyCmd( ClientData clientData, Tcl_Interp *interp,
              int argc, char *argv[] )
{
    tclMatrix *w;

    w = Tcl_GetMatrixPtr( interp, argv[1] );
    ...
```

To learn about what else you can do with the matrix once inside compiled code, read `tclMatrix.h` to learn the definition of the `tclMatrix` structure, and see the examples in files like `tclAPI.c` which show many various uses of the Tcl matrix.

Using Tcl Matrices from C++

Using a Tcl matrix from C++ is very much like using it from C, except that `tclMatrix.h` contains some C++ wrapper classes which are somewhat more convenient than using the indexing macros which one has to use in C. For example, here is a tiny snippet from one of the authors codes in which Tcl matrices are passed in from Tcl to a C++ routine which is supposed to fill them in with values from some matrices used in the compiled side of the code:

```
...
if (item == "vertex_coords") {
    tclMatrix *matxg = Tcl_GetMatrixPtr( interp, argv[1] );
    tclMatrix *matyg = Tcl_GetMatrixPtr( interp, argv[2] );

    Mat2<float> xg(ncu, ncv), yg(ncu, ncv);
    cg->Get_Vertex_Coords( xg, yg );

    TclMatFloat txg( matxg ), tyg( matyg );

    for( i=0; i < ncu; i++ )
        for( j=0; j < ncv; j++ ) {
            txg(i,j) = xg(i,j);
            tyg(i,j) = yg(i,j);
        }
```

There are other things you can do too, see the definitions of the `TclMatFloat` and `TclMatInt` classes in `tclMatrix.h`.

Extending the Tcl Matrix facility

The Tcl matrix facility provides creation, indexing, and information gathering facilities. However, considering the scientifically inclined PLplot user base, it is clear that some users will demand more. Consequently there is a mechanism for augmenting the Tcl matrix facility with your own, user defined, extension subcommands. Consider `xtk04.c`. In this extended wish, we want to be able to determine the minimum and maximum values stored in a matrix. Doing this in Tcl would involve nested loops, which in Tcl would be prohibitively slow. We could register a Tcl extension command to do it, but since the only sensible data for such a command would be a Tcl matrix, it seems nice to provide this facility as an actual subcommand of the matrix. However, the PLplot maintainers cannot foresee every need, so a mechanism is provided to register subcommands for use with matrix objects.

The way to register matrix extension subcommands is to call `Tcl_MatrixInstallXtnsn`:

```
typedef int (*tclMatrixXtnsnProc) ( tclMatrix *pm, Tcl_Interp *interp,
                                    int argc, char *argv[] );

int Tcl_MatrixInstallXtnsn( char *cmd, tclMatrixXtnsnProc proc );
```

In other words, make a function for handling the matrix extension subcommand, with the same function signature (prototype) as `tclMatrixXtnsnProc`, and register the subcommand name along with the function pointer. For example, xtk04.c has:

```
int mat_max( tclMatrix *pm, Tcl_Interp *interp,
             int argc, char *argv[] )
{
```

```
    float max = pm->fdata[0];
    int i;
    for( i=1; i < pm->len; i++ )
        if (pm->fdata[i] > max)
            max = pm->fdata[i];

    sprintf( interp->result, "%f", max );
    return TCL_OK;
}

int mat_min( tclMatrix *pm, Tcl_Interp *interp,
             int argc, char *argv[] )
{
    float min = pm->fdata[0];
    int i;
    for( i=1; i < pm->len; i++ )
        if (pm->fdata[i] < min)
            min = pm->fdata[i];

    sprintf( interp->result, "%f", min );
    return TCL_OK;
}
```

Then, inside the application initialization function (`Tcl_AppInit()` to long time Tcl users):

```
Tcl_MatrixInstallXtnsn( "max", mat_max );
Tcl_MatrixInstallXtnsn( "min", mat_min );
```

Then we can do things like:

```
dino 65: xtk04
% matrix x f 4 = {1, 2, 3, 1.5}
% x min
1.000000
% x max
3.000000
```

Your imagination is your only limit for what you can do with this. You could add an FFT subcommand, matrix math, BLAS, whatever.

Contouring and Shading from Tcl

Contouring and shading has traditionally been one of the messier things to do in PLplot. The C API has many parameters, with complex setup and tear down properties. Of special concern is that some of the parameters do not have a natural representation in script languages like Tcl. In this section we describe how the Tcl interface to these facilities is provided, and how to use it.

Drawing a Contour Plot from Tcl

By way of reference, the primary C function call for contouring is:

```
void plcont( PLFLT **f, PLINT nx, PLINT ny, PLINT kx, PLINT lx,
             PLINT ky, PLINT ly, PLFLT *clevel, PLINT nlevel,
             void (*pltr) (PLFLT, PLFLT, PLFLT *, PLFLT *, PLPointer),
             PLPointer pltr_data);
```

This is a fairly complex argument list, and so for this function (and for plshade, described below) we dispense with trying to exactly mirror the C API, and just concentrate on capturing the functionality

within a Tcl context. To begin with, the data is provided through a 2-d Tcl matrix. The Tcl matrix carries along its size information with it, so `nx` and `ny` are no longer needed. The `kx`, `lx`, `ky` and `ly` variables are potentially still useful for plotting a subdomain of the full data set, so they may be specified in the natural way, but we make this optional since they are frequently not used to convey anything more than what could be inferred from `nx` and `ny`. However, to simplify processing, they must be supplied or omitted as a set (all of them, or none of them). `clevel` is supplied as a 1-d Tcl matrix, and so `nlevel` can be omitted.

Finally, we have no way to support function pointers from Tcl, so instead we provide token based support for accessing the three coordinate transformation routines which are provided by PLplot, and which many PLplot users use. There are thus three courses of action:

Provide no pltr specification. In this case, `pltr0` is used by default.

Specify `pltr1 x y` where x and y are 1-d Tcl matrices. In this case `pltr1` will be used, and the 1-d arrays which it needs will be supplied from the Tcl matrices `x` and `y`.

Specify `pltr2 x y` where x and y are 2-d Tcl matrices. In this case `pltr2` will be used, and the 2-d arrays which it needs will be supplied from the Tcl matrices `x` and `y`.

Now, there can be no question that this is both more concise and less powerful than what you could get in C. The loss of the ability to provide a user specified transformation function is regrettable. If you really do need that functionality, you will have to implement your own Tcl extension command to do pretty much the same thing as the provided Tcl extension command `plcont` (which is in `tclAPI.c` in function `plcontCmd()`), except specify the C transformation function of your choice.

However, that having been said, we recognize that one common use for this capability is to provide a special version of `pltr2` which knows how to implement a periodic boundary condition, so that polar plots, for example, can be implemented cleanly. That is, if you want to draw contours of a polar data set defined on a 64 x 64 grid, ensuring that contour lines would actually go all the way around the origin rather than breaking off like a silly pacman figure, then you had basically two choices in C. You could copy the data to a 65 x 64 grid, and replicate one row of data into the spare slot, and then plot the larger data set (taking care to replicate the coordinate arrays you passed to pltr2 in the same way), *or* you could make a special version of `pltr2` which would understand that one of the coordinates was wrapped, and perform transformations accordingly without actually making you replicate the data.

Since the former option is ugly in general, and hard to do in Tcl in particular, and since the second option is even more difficult to do in Tcl (requiring you do make a special Tcl extension command as described above), we provide special, explicit support for this common activity. This is provided through the use of a new, optional parameter `wrap` which may be specified as the last parameter to the Tcl command, only if you are using `pltr2`. Supplying `1` will wrap in the first coordinate, `2` will wrap in the second coordinate.

The resultant Tcl command is:

```
plcont f [kx lx ky ly] clev [pltr x y] [wrap]
```

Note that the brackets here are used to signify optional arguments, *not* to represent Tcl command substitution!

The Tcl demo `x09.tcl` provides examples of all the capabilities of this interface to contouring from Tcl. Note in particular, `x09_polar` which does a polar contour without doing anything complicated in the way of setup, and without getting a pacman as the output.

Drawing a Shaded Plot from Tcl

The Tcl interface to shading works very much like the one for contouring. The command is:

```
plshade z xmin xmax ymin ymax \
    sh_min sh_max sh_cmap sh_color sh_width \
    min_col min_wid max_col max_wid \
    rect [pltr x y] [wrap]
```

where `nx` and `ny` were dropped since they are inferred from the Tcl matrix `z`, `defined` was dropped since it isn't supported anyway, and `plfill` was dropped since it was the only valid choice anyway. The `pltr` spec and `wrap` work exactly as described for the Tcl `plcont` described above.

The Tcl demo `x16.tcl` contains extensive demonstrations of use, including a shaded polar plot which connects in the desirable way without requiring special data preparation, again just like for `plcont` described previously.

Understanding the Performance Characteristics of Tcl

Newcomers to Tcl, and detractors (read, "proponents of other paradigms") often do not have a clear (newcomers) or truthful (detractors) perspective on Tcl performance. In this section we try to convey a little orientation which may be helpful in working with the PLplot Tcl interface.

"Tcl is slow!" "Yeah, so what?"

Debates of this form frequently completely miss the point. Yes, Tcl is definitely slow. It is fundamentally a string processing language, is interpreted, and must perform substitutions and so forth on a continual basis. All of that takes time. Think milliseconds instead of microseconds for comparing Tcl code to equivalent C code. On the other hand, this does not have to be problematic, even for time critical (interactive) applications, if the division of labor is done correctly. Even in an interactive program, you can use Tcl fairly extensively for high level control type operations, as long as you do the real work in a compiled Tcl command procedure. If the high level control code is slow, so what? So it takes 100 milliseconds over the life the process, as compared to the 100 microseconds it could have taken if it were in C. Big deal. On an absolute time scale, measured in units meaningful to humans, it's just not a lot of time.

The problem comes when you try to do too much in Tcl. For instance, an interactive process should not be trying to evaluate a mathematical expression inside a doubly nested loop structure, if performance is going to be a concern.

Case in point: Compare x16.tcl to x16c.c. The code looks very similar, and the output looks very similar. What is not so similar is the execution time. The Tcl code, which sets up the data entirely in Tcl, takes a while to do so. On the other hand, the actual plotting of the data proceeds at a rate which is effectively indistinguishable from that of the compiled example. On human time scales, the difference is not meaningful. Conclusion: If the computation of the data arrays could be moved to compiled code, the two programs would have performance close enough to identical that it really wouldn't be an issue. We left the Tcl demos coded in Tcl for two reasons. First because they provide some examples and tests of the use of the Tcl Matrix extension, and secondly because they allow the Tcl demos to be coded entirely in Tcl, without requiring special customized extended shells for each one of them. They are not, however, a good example of you should do things in practice.

Now look at `tk04` and `xtk04.c`, you will see that if the data is computed in compiled code, and shuffled into the Tcl matrix and then plotted from Tcl, the performance is fine. Almost all the time is spent in plshade, in compiled code. The time taken to do the small amount of Tcl processing involved with plotting is dwarfed by the time spent doing the actual drawing in C. So using Tcl cost almost nothing in this case.

So, the point is, do your heavy numerics in a compiled language, and feel free to use Tcl for the plotting, if you want to. You can of course mix it up so that some plotting is done from Tcl and some from a compiled language.

Chapter 10. 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 ...

Introduction to Tcl

The Tool Command Language, or just Tcl (pronounced "tickle") is an embeddable 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.

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.

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 procedures which effectively become new keywords.

But the real power of this approach is to add new commands to the interpreter 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 interpreter 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.

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`[1]. 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 anonymous **ftp** site for Tcl is `harbor.ecn.purdue.edu:/pub/tcl`[2]. 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 driver in PLplot, you will need Tcl-8.2 and Tk-8.2 (or higher versions). Additionally, in order to use the extended WISH paradigm (described below) you will need iTcl-3.1 (or a higher version).

However, you will quite likely find Tcl/Tk to be very addictive, and the great plethora of add-ons available at `harbor` will 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 interpreter. The benefits of this approach will become apparent as you gain experience with Tcl and Tk.

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 widgets 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 driver 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 utilities ranging from mail to info, to SQL, to news, etc. The list is endless and growing fast...

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 oriented 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.

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 facilities 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,

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 graphical 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". 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...

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 interpreter, 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 miscellany 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 deleted then 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 interpreter, 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].

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.

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 implement X Windows user interfaces to control and utilize the facilities made available in your extensions.

Effectively this just comes down to writing 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 when your programming convenience is more important than raw speed.

Notes

1. ftp://sprite.berkeley.edu/tcl
2. ftp://harbor.ecn.purdue.edu/pub/tcl

Chapter 11. Constructing Graphical Interfaces with PLplot

This chapter presents some ideas on how to use PLplot in conjunction with Tcl, Tk, and [incr Tcl] to construct graphical interfaces to scientific codes. blah blah

Chapter 12. Using PLplot from Perl

There are no proper bindings for the Perl language delivered with the PLplot sources. However, a PLplot interface has been added to the Perl Data Language (PDL) since version 2.4.0. If the PLplot library is installed in the system, it is automatically detected by the PDL configuration script, such that PLplot support for PDL should work out of the box.

For further informations see the PDL homepage[1].

Notes

1. http://pdl.perl.org

Chapter 13. Using PLplot from Python

NEEDS DOCUMENTATION, but here is the short story. We currently (February, 2001) have switched to dynamic loading of plplot following the generic method given in the python documentation. Most (???) of the PLplot common API has been implemented. (For a complete list see plmodules.c and plmodules2.c). With this dynamic method all the xw??.py examples work fine and should be consulted for the best way to use PLplot from python. You may have to set PYTHONPATH to the path where plmodule.so is located (or eventually installed). For more information see examples/python/README

pytkdemo and the x??.py examples it loads use the plframe widget. Thus, this method does not currently work under dynamic loading. They have only worked in the past using the static method with much hacking and rebuilding of python itself. We plan to try dynamic loading of all of PLplot (not just the plmodule.c and plmodule2.c wrappers) including plframe (or a python-variant of this widget) into python at some future date to see whether it is possible to get pytkdemo and the x??.py examples working under dynamic loading, but only the individual stand-alone xw??.py demos work at the moment.

IV. Reference

Chapter 14. Bibliography

These articles are descriptions of PLplot itself or else scientific publications whose figures were generated with PLplot.

References

Furnish G., "Das Graphikpaket PLplot (in German) (http://www.linux-magazin.de/ausgabe/1996/12/Plplot/plplot.html)", *Linux Magazin*, 1996 December

Furnish G., Horton W., Kishimoto Y., LeBrun M., Tajima T., "Global Gyrokinetic Simulation of Tokamak Transport", *Physics of Plasmas*, 6, 1, 1999

Irwin A.W., Fukushima T., "A Numerical Time Ephemeris of the Earth", *Astronomy and Astrophysics*, 348, 642, 1999

LeBrun M.J., Tajima T., Gray M., Furnish G., Horton W., "Toroidal Effects on Drift-Wave Turbulence", *Physics of Fluids*, B5, 752, 1993

Chapter 15. The Common API for PLplot

The purpose of this chapter is to document the API for every C function in PLplot that should have a counterpart in other PLplot language bindings such as Fortran. These common API routines have a special "c " prefix name assigned to them in `plplot.h`. This common API between the various languages constitutes the most important part of the PLplot API that programmers need to know. Additional PLplot API specialized for each language binding is documented in Chapter 16 and subsequent chapters.

All common API functions of the current CVS HEAD are listed here with their arguments. All functions (other than the deprecated ones) have at a short description, and all parameters are documented.

`pl_setcontlabelformat`: Set format of numerical label for contours

```
pl_setcontlabelformat (lexp, sigdig);
```

Set format of numerical label for contours.

`lexp` (`PLINT`, input)

If the contour numerical label is greater than 10 (lexp) or less than 10 (-lexp), then the exponential format is used. Default value of lexp is 4.

`sigdig` (`PLINT`, input)

Number of significant digits. Default value is 2.

`pl_setcontlabelparam`: Set parameters of contour labelling other than format of numerical label

```
pl_setcontlabelparam (offset, size, spacing, active);
```

Set parameters of contour labelling other than those handled by `pl_setcontlabelformat`.

`offset` (`PLFLT`, input)

Offset of label from contour line (if set to 0.0, labels are printed on the lines). Default value is 0.006.

`size` (`PLFLT`, input)

Font height for contour labels (normalized). Default value is 0.3.

`spacing` (`PLFLT`, input)

Spacing parameter for contour labels. Default value is 0.1.

`active` (`PLINT`, input)

Activate labels. Set to 1 if you want contour labels on. Default is off (0).

`pladv`: Advance the (sub-)page

```
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 pleop followed by plbop 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.

plaxes: Draw a box with axes, etc. with arbitrary origin

```
plaxes (x0, y0, xopt, xtick, nxsub, yopt, ytick, nysub);
```

Draws a box around the currently defined viewport with arbitrary world-coordinate origin specified by *x0* and *y0* and labels it with world coordinate values appropriate to the window. Thus plaxes 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.

x0 (PLFLT, input)

World X coordinate of origin.

y0 (PLFLT, input)

World Y coordinate of origin.

xopt (const 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.

f: Always use fixed point numeric labels.

g: Draws a grid at the major tick interval.

h: Draws a grid at the minor 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 (`const 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 (`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.

`plbin`: Plot a histogram from binned data

```
plbin (nbin, x, y, center);
```

Plots a histogram consisting of *nbin* 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*`=0`, *x*`[i]` is the left-hand edge of the `i`'th bin, and if *center*`=1`, 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.

center (PLINT, input)

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

plbop: Begin a new page

```
plbop ();
```

Begins a new page. For a file driver, the output file is opened if necessary. Advancing the page via pleop and plbop is useful when a page break is desired at a particular point when plotting to subpages. Another use for pleop and plbop 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 pleop (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.

plbox: Draw a box with axes, etc

```
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 (const 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.

f: Always use fixed point numeric labels.

g: Draws a grid at the major tick interval.

h: Draws a grid at the minor 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 (`const 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 (`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.

plbox3: Draw a box with axes, etc, in 3-d

```
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 the Section called *Three Dimensional Surface Plots* in Chapter 3.

xopt (`const 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.

`f`: Always use fixed point numeric labels.

`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 (const 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 (const char *, input)

Pointer to character string specifying options for the y axis. The string is interpreted in the same way as *xopt*.

ylabel (const 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 (const 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.

f: Always use fixed point numeric labels.

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 vertical axis.

n: Writes numeric labels at major tick intervals on the left-hand vertical 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 (`const 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.

plcalc_world: Calculate world coordinates and corresponding window index from relative device coordinates

```
plcalc_world (rx, ry, wx, wy, window);
```

Calculate world coordinates, *wx* and *wy*, and corresponding *window* index from relative device coordinates, *rx* and *ry*.

rx (`PLFLT`, input)

Input relative device coordinate (ranging from 0. to 1.) for the x coordinate.

ry (`PLFLT`, input)

Input relative device coordinate (ranging from 0. to 1.) for the y coordinate.

wx (`PLFLT *`, output)

Pointer to the returned world coordinate for x corresponding to the relative device coordinates *rx* and *ry*.

wy (`PLFLT *`, output)

Pointer to the returned world coordinate for y corresponding to the relative device coordinates *rx* and *ry*.

window (`PLINT *`, output)

Pointer to the returned last defined window index that corresponds to the input relative device coordinates (and the returned world coordinates). To give some background on the window index, for each page the initial window index is set to zero, and each time `plwind` is called within the page, world and device coordinates are stored for the window and the window index is incremented. Thus, for a simple page layout with non-overlapping viewports and one window per viewport, *window* corresponds to the viewport index (in the order which the viewport/windows were created) of the only viewport/window corresponding to *rx* and *ry*. However, for more complicated layouts with potentially overlapping viewports and possibly more than one window (set of world coordinates) per viewport, *window* and the corresponding output world coordinates corresponds to the last window created that fulfils the criterion that the relative device coordinates are inside it. Finally, in all cases where the input relative device coordinates are not inside any viewport/window, then *window* is set to -1.

plclear: Clear current (sub)page

```
plclear ();
```

Clears the current page, effectively erasing everything that have been drawn. This command only works with interactive drivers; if the driver does not support this, the page is filled with the background color in use. If the current page is divided into subpages, only the current subpage is erased. The nth subpage can be selected with pladv(n).

plclr: Eject current page

```
plclr ();
```

Deprecated. Use the new name, pleop, for this function instead.

plcol: Set color

```
plcol (color);
```

Deprecated. Use the new name, plcol0, for this function instead.

color (PLINT, input)

See plcol0.

plcol0: Set color, map0

```
plcol0 (color);
```

Sets the color for color map0 (see the Section called *Color Map0* in Chapter 3).

color (PLINT, input)

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

0 black (default background)
1 red (default foreground)
2 yellow
3 green
4 aquamarine
5 pink
6 wheat
7 grey
8 brown
9 blue
10 BlueViolet
11 cyan
12 turquoise
13 magenta

14 salmon
15 white

Use plscmap0 to change the entire map0 color palette and plscol0 to change an individual color in the map0 color palette.

plcol1: Set color, map1

```
plcol1 (col1);
```

Sets the color for color map1 (see the Section called *Color Map1* in Chapter 3).

col1 (PLFLT, input)

This value must be in the range from 0. to 1. and is mapped to color using the continuous map1 color palette which by default ranges from blue to the background color to red. The map1 palette can also be straightforwardly changed by the user with plscmap1 or plscmap1l.

plcont: Contour plot

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

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 pointed to by *pltr* with a pointer *pltr_data* for additional data required by the transformation routine is used to map indices within the array to the world coordinates. See the following discussion of the arguments and the Section called *Contour and Shade Plots* in Chapter 3 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 indices to consider.

ky, *ly* (PLINT, input)

Range of y indices 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 (*) (PLFLT, PLFLT, PLFLT *, PLFLT *, PLPointer) , input)

Pointer to function that defines transformation between indices in array *z* and the world coordinates (C only). Transformation functions are provided in the PLplot library: pltr0 for identity mapping, and pltr1 and pltr2 for arbitrary mappings respectively defined by one- and two-dimensional arrays. In addition, user-supplied routines for the transformation can be used as well. Examples of all of these

approaches are given in the Section called *Contour Plots from C* in Chapter 3. The transformation function should have the form given by any of `pltr0`, `pltr1`, or `pltr2`.

`pltr_data` (`PLPointer`, input)

Extra parameter to help pass information to `pltr0`, `pltr1`, `pltr2`, or whatever routine that is externally supplied.

plcpstrm: Copy state parameters from the reference stream to the current stream

```
plcpstrm (iplsr, flags);
```

Copies state parameters from the reference stream to the current stream. Tell driver interface to map device coordinates unless *`flags`* `== 1`.

This function is used for making save files of selected plots (e.g. from the TK driver). After initializing, you can get a copy of the current plot to the specified device by switching to this stream and issuing a `plcpstrm` and a `plreplot`, with calls to `plbop` and `pleop` as appropriate. The plot buffer must have previously been enabled (done automatically by some display drivers, such as X).

`iplsr` (`PLINT`, input)

Number of reference stream.

`flags` (`PLINT`, input)

If *`flags`* is set to 1 the device coordinates are *not* copied from the reference to current stream.

plend: End plotting session

```
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: End plotting session for current stream

```
plend1 ();
```

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

plenv0: Same as plenv() but if in multiplot mode does not advance the subpage, instead clears it.

```
plenv0 (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. `plenv0` leaves enough room around most graphs for axis labels and a title. When these defaults are not suitable, use the individual routines `plvpas`, `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)

Controls how the axes will be scaled:

`-1`: the scales will not be set, the user must set up the scale before calling plenv0() using plsvpa(), plvasp() or other.

`0`: the x and y axes are scaled independently to use as much of the screen as possible.

`1`: the scales of the x and y axes are made equal.

`2`: the axis of the x and y axes are made equal, and the plot box will be square.

axis (PLINT, input)

Controls drawing of the box around the plot:

`-2`: draw no box, no tick marks, no numeric tick labels, no axes.

`-1`: draw box only.

`0`: draw box, ticks, and numeric tick labels.

`1`: also draw coordinate axes at `x=0` and `y=0`.

`2`: also draw a grid at major tick positions in both coordinates.

`3`: also draw a grid at minor tick positions in both coordinates.

`10`: same as 0 except logarithmic `x` tick marks. (The `x` data have to be converted to logarithms separately.)

`11`: same as 1 except logarithmic `x` tick marks. (The `x` data have to be converted to logarithms separately.)

`12`: same as 2 except logarithmic `x` tick marks. (The `x` data have to be converted to logarithms separately.)

`13`: same as 3 except logarithmic `x` tick marks. (The `x` data have to be converted to logarithms separately.)

`20`: same as 0 except logarithmic `y` tick marks. (The `y` data have to be converted to logarithms separately.)

`21`: same as 1 except logarithmic `y` tick marks. (The `y` data have to be converted to logarithms separately.)

22: same as 2 except logarithmic y tick marks. (The y data have to be converted to logarithms separately.)

23: same as 3 except logarithmic y tick marks. (The y data have to be converted to logarithms separately.)

30: same as 0 except logarithmic x and y tick marks. (The x and y data have to be converted to logarithms separately.)

31: same as 1 except logarithmic x and y tick marks. (The x and y data have to be converted to logarithms separately.)

32: same as 2 except logarithmic x and y tick marks. (The x and y data have to be converted to logarithms separately.)

33: same as 3 except logarithmic x and y tick marks. (The x and y data have to be converted to logarithms separately.)

plenv: Set up standard window and draw box

```
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 plvpas, 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)

Controls how the axes will be scaled:

-1: the scales will not be set, the user must set up the scale before calling plenv() using plsvpa(), plvasp() or other.

0: the x and y axes are scaled independently to use as much of the screen as possible.

1: the scales of the x and y axes are made equal.

2: the axis of the x and y axes are made equal, and the plot box will be square.

axis (PLINT, input)

Controls drawing of the box around the plot:

-2: draw no box, no tick marks, no numeric tick labels, no axes.

`-1`: draw box only.

`0`: draw box, ticks, and numeric tick labels.

`1`: also draw coordinate axes at `x=0` and `y=0`.

`2`: also draw a grid at major tick positions in both coordinates.

`3`: also draw a grid at minor tick positions in both coordinates.

`10`: same as 0 except logarithmic `x` tick marks. (The `x` data have to be converted to logarithms separately.)

`11`: same as 1 except logarithmic `x` tick marks. (The `x` data have to be converted to logarithms separately.)

`12`: same as 2 except logarithmic `x` tick marks. (The `x` data have to be converted to logarithms separately.)

`13`: same as 3 except logarithmic `x` tick marks. (The `x` data have to be converted to logarithms separately.)

`20`: same as 0 except logarithmic `y` tick marks. (The `y` data have to be converted to logarithms separately.)

`21`: same as 1 except logarithmic `y` tick marks. (The `y` data have to be converted to logarithms separately.)

`22`: same as 2 except logarithmic `y` tick marks. (The `y` data have to be converted to logarithms separately.)

`23`: same as 3 except logarithmic `y` tick marks. (The `y` data have to be converted to logarithms separately.)

`30`: same as 0 except logarithmic `x` and `y` tick marks. (The `x` and `y` data have to be converted to logarithms separately.)

`31`: same as 1 except logarithmic `x` and `y` tick marks. (The `x` and `y` data have to be converted to logarithms separately.)

`32`: same as 2 except logarithmic `x` and `y` tick marks. (The `x` and `y` data have to be converted to logarithms separately.)

`33`: same as 3 except logarithmic `x` and `y` tick marks. (The `x` and `y` data have to be converted to logarithms separately.)

`pleop`: Eject current page

```
pleop ();
```

Clears the graphics screen of an interactive device, or ejects a page on a plotter. See `plbop` for more information.

`plerrx`: Draw x error bar

```
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: Draw y 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.

plfamadv: Advance to the next family file on the next new page

```
plfamadv ();
```

Advance to the next family file on the next new page.

plfill: Area fill

```
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.

`plfill3`: Area fill in 3D

```
plfill3 (n, x, y, z);
```

Fills the 3D polygon defined by the *n* points in the *x*, *y*, and *z* arrays 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*, *y*, and *z* then `plfill3` 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.

z (`PLFLT *`, input)

Pointer to array with z coordinates of vertices.

`plflush`: Flushes the output stream

```
plflush ();
```

Flushes the output stream. Use sparingly, if at all.

`plfont`: Set character font

```
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: Load character 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

plgchr: Get character default height and current (scaled) height

```
plgchr (p_def, p_ht);
```

Get character default height and current (scaled) height.

p_def (PLFLT *, output)

Pointer to default character height.

p_ht (PLFLT *, output)

Pointer to current (scaled) character height.

plgcol0: Returns 8-bit RGB values for given color from color map0

```
plgcol0 (icol0, r, g, b);
```

Returns 8-bit RGB values for given color from color map0 (see the Section called *Color Map0* in Chapter 3). Values are negative if an invalid color id is given.

icol0 (PLINT, input)

Index of desired cmap0 color.

r (PLINT *, output)

Pointer to 8-bit red value.

g (PLINT *, output)

Pointer to 8-bit green value.

b (PLINT *, output)

Pointer to 8-bit blue value.

plgcolbg: Returns the background color (cmap0[0]) by 8-bit RGB value

```
plgcolbg (r, g, b);
```

Returns the background color (cmap0[0]) by 8-bit RGB value.

r (PLINT *, output)

Pointer to an unsigned 8-bit integer (0-255) representing the degree of red in the color.

g (PLINT *, output)

Pointer to an unsigned 8-bit integer (0-255) representing the degree of green in the color.

b (PLINT *, output)

Pointer to an unsigned 8-bit integer (0-255) representing the degree of blue in the color.

plgcompression: Get the current device-compression setting

```
plgcompression (compression);
```

Get the current device-compression setting. This parameter is only used for drivers that provide compression.

compression (PLINT *, output)

Pointer to a variable to be filled with the current device-compression setting.

plgdev: Get the current device (keyword) name

```
plgdev (p_dev);
```

Get the current device (keyword) name. Note: you *must* have allocated space for this (80 characters is safe).

p_dev (char *, output)

Pointer to device (keyword) name string.

plgdidev: Get parameters that define current device-space window

```
plgdidev (p_mar, p_aspect, p_jx, p_jy);
```

Get relative margin width, aspect ratio, and relative justification that define current device-space window. If plsdidev has not been called the default values pointed to by *p_mar*, *p_aspect*, *p_jx*, and *p_jy* will all be 0.

p_mar (PLFLT *, output)

Pointer to relative margin width.

p_aspect (PLFLT *, output)

Pointer to aspect ratio.

p_jx (PLFLT *, output)

Pointer to relative justification in x.

p_jy (PLFLT *, output)

Pointer to relative justification in y.

plgdiori: Get plot orientation

```
plgdiori (p_rot);
```

Get plot orientation parameter which is multiplied by 90 to obtain the angle of rotation. Note, arbitrary rotation parameters such as 0.2 (corresponding to 18) are possible, but the usual values for the rotation parameter are 0., 1., 2., and 3. corresponding to 0 (landscape mode), 90 (portrait mode), 180 (seascape mode), and 270 (upside-down mode). If plsdiori has not been called the default value pointed to by *p_rot* will be 0.

p_rot (PLFLT *, output)

Pointer to orientation parameter.

plgdiplt: Get parameters that define current plot-space window

```
plgdiplt (p_xmin, p_ymin, p_xmax, p_ymax);
```

Get relative minima and maxima that define current plot-space window. If plsdiplt has not been called the default values pointed to by *p_xmin*, *p_ymin*, *p_xmax*, and *p_ymax* will be 0., 0., 1., and 1.

p_xmin (PLFLT *, output)

Pointer to relative minimum in x.

p_ymin (PLFLT *, output)

Pointer to relative minimum in y.

p_xmax (PLFLT *, output)

Pointer to relative maximum in x.

p_ymax (PLFLT *, output)

Pointer to relative maximum in y.

plgfam: Get family file parameters

```
plgfam (fam, num, bmax);
```

Gets information about current family file, if familying is enabled. See the Section called *Family File Output* in Chapter 3 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.

plgfnam: Get output file name

```
plgfnam (fnam);
```

Gets the current output file name, if applicable.

fnam (char *, output)

Pointer to file name string.

plglevel: Get the (current) run level

```
plglevel (p_level);
```

Get the (current) run level. Valid settings are:

0, uninitialized

1, initialized

2, viewport defined

3, world coords defined

p_level (PLINT *, output)

Pointer to the run level.

plgpage: Get page parameters

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

Gets the current page configuration.

xp (PLFLT *, output)

Pointer to number of pixels in x.

yp (PLFLT *, 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: Switch to graphics screen

```
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.

plgriddata: Grid data from irregularly sampled data

```
plggriddata (x, y, z, npts, xg, nptsx, yg, nptsy, zg, type, data);
```

Real world data is frequently irregularly sampled, but all PLplot 3D plots require data placed in a uniform grid. This function takes irregularly sampled data from three input arrays `x[npts]`, `y[npts]`, and `z[npts]`, reads the desired grid location from input arrays `xg[nptsx]` and `yg[nptsy]`, and returns the

gridded data into output array `zg[nptsx][nptsy]`. The algorithm used to grid the data is specified with the argument `type` which can have one parameter specified in argument `data`.

x (`PLFLT *`, input)

The input `x` array.

y (`PLFLT *`, input)

The input `y` array.

z (`PLFLT *`, input)

The input `z` array. Each triple `x[i]`, `y[i]`, `z[i]` represents one data sample coordinates.

npts (`PLINT`, input)

The number of data samples in the `x`, `y` and `z` arrays.

xg (`PLFLT *`, input)

The input array that specifies the grid spacing in the x direction. Usually `xg` has `nptsx` equaly spaced values from the mininum to the maximum values of the `x` input array.

nptsx (`PLINT`, input)

The number of points in the `xg` array.

yg (`PLFLT *`, input)

The input array that specifies the grid spacing in the y direction. Similar to the `xg` parameter.

nptsy (`PLINT`, input)

The number of points in the `yg` array.

sg (`PLFLT **`, output)

The output array, where data lies in the regular grid specified by `xg` and `yg`. the `zg` array must exists or be allocated by the user prior to the calling, and must have dimension `zg[nptsx][xptsy]`.

type (`PLINT`, input)

The type of gridding algorithm to use, which can be:

`GRID_CSA`: Bivariate Cubic Spline approximation

`GRID_DTLI`: Delaunay Triangulation Linear Interpolation

`GRID_NNI`: Natural Neighbors Interpolation

`GRID_NNIDW`: Nearest Neighbors Inverse Distance Weighted

`GRID_NNLI`: Nearest Neighbors Linear Interpolation

`GRID_NNAIDW`: Nearest Neighbors Around Inverse Distance Weighted

For details on the algorithm read the source file `plgridd.c`.

data (`PLFLT`, input)

Some gridding algorithms require extra data, which can be specified through this argument. Currently, for algoritm:

`GRID_NNIDW`, `data` specifies the number of neighbors to use, the lower the value, the noisier (more local) the approximation is.

`GRID_NNLI`, `data` specifies what a thin triangle is, in the range [1. .. 2.]. High values enable the usage of very thin triangles for interpolation, possibly resulting in error in the approximation.

GRID_NNI, only weights greater then data will be accepted. If 0, all weigths will be accepted.

plgspa: Get current subpage parameters

```
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: Get current stream number

```
plgstrm (strm);
```

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

strm (PLINT *, output)

Pointer to current stream value.

plgver: Get the current library version number

```
plgver (p_ver);
```

Get the current library version number. Note: you *must* have allocated space for this (80 characters is safe).

p_ver (char *, output)

Pointer to the current library version number.

plgvpd: Get viewport limits in normalized device coordinates

```
plgvpd (p_xmin , p_xmax , p_ymin , p_ymax );
```

Get viewport limits in normalized device coordinates.

p_xmin (PLFLT *, output)

Lower viewport limit of the normalized device coordinate in x.

p_xmax (PLFLT *, output)

Upper viewport limit of the normalized device coordinate in x.

p_ymin (PLFLT *, output)

Lower viewport limit of the normalized device coordinate in y.

p_ymax (PLFLT *, output)

Upper viewport limit of the normalized device coordinate in y.

plgvpw: Get viewport limits in world coordinates

```
plgvpw (p_xmin , p_xmax , p_ymin , p_ymax );
```

Get viewport limits in world coordinates.

p_xmin (PLFLT *, output)

Lower viewport limit of the world coordinate in x.

p_xmax (PLFLT *, output)

Upper viewport limit of the world coordinate in x.

p_ymin (PLFLT *, output)

Lower viewport limit of the world coordinate in y.

p_ymax (PLFLT *, output)

Upper viewport limit of the world coordinate in y.

plgxax: Get x axis parameters

```
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 this routine should only be called *after* the call to plbox (or plbox3) is complete. See the Section called *Annotating the Viewport* in Chapter 3 for more information.

digmax (PLINT *, output)

Pointer to variable with the maximum number of digits for the x axis. If nonzero, the printed label has been 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: Get y axis parameters

```
plgyax (digmax, digits);
```

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

digmax (PLINT *, output)

Pointer to variable with the maximum number of digits for the y axis. If nonzero, the printed label has been 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 (y axis) from the last plot.

plgzax: Get z axis parameters

```
plgzax (digmax, digits);
```

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

digmax (PLINT *, output)

Pointer to variable with the maximum number of digits for the z axis. If nonzero, the printed label has been 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 (z axis) from the last plot.

plhist: Plot a histogram from unbinned data

```
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.

plhls: Set current color by HLS

```
plhls (h, l, s);
```

Set current color by hue, lightness, and saturation. Convert hls color coordinates to rgb, then call plrgb. Do *not* use this. Only retained for backward compatibility

h (PLFLT, input)

NEEDS DOCUMENTATION

l (PLFLT, input)

NEEDS DOCUMENTATION

s (PLFLT, input)

NEEDS DOCUMENTATION

plinit: Initialize PLplot

```
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, the plsetopt function, 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: Draw a line between two points

```
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: Simple routine to write labels

```
pllab (xlabel, ylabel, tlabel);
```

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

xlabel (const char *, input)

Label for horizontal axis.

ylabel (const char *, input)

Label for vertical axis.

tlabel (const char *, input)

Title of graph.

pllightsource: Sets the 3D position of the light source

```
pllightsource (x, y, z);
```

Sets the 3D position of the light source for use with `plsurf3d`.

x (PLFLT, input)

X-coordinate of the light source.

y (PLFLT, input)

Y-coordinate of the light source.

z (PLFLT, input)

Z-coordinate of the light source.

plline: Draw a line

```
plline (n, x, y);
```

Draws line defined by *n* points in *x* and *y*.

n (PLINT, input)

Number of points defining line.

x (PLFLT *, input)

Pointer to array with x coordinates of points.

y (PLFLT *, input)

Pointer to array with y coordinates of points.

plline3: Draw a line in 3 space

```
plline3 (n, x, y, z);
```

Draws line in 3 space defined by *n* points in *x*, *y*, and *z*. You must first set up the viewport, the 2d viewing window (in world coordinates), and the 3d normalized coordinate box. See `x18c.c` for more info.

n (PLINT, input)

Number of points defining line.

x (PLFLT *, input)

Pointer to array with x coordinates of points.

y (PLFLT *, input)

Pointer to array with y coordinates of points.

z (PLFLT *, input)

Pointer to array with z coordinates of points.

pllsty: Select line style

```
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: Plot surface mesh

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

Plots a surface mesh within the environment set up by plw3d. The surface is defined by the two-dimensional array *z*[*nx*][*ny*], 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 the Section called *Three Dimensional Surface Plots* in Chapter 3.

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:

opt=DRAW_LINEX: Lines are drawn showing *z* as a function of *x* for each value of *y*[j].

opt=DRAW_LINEY: Lines are drawn showing *z* as a function of *y* for each value of *x*[i].

opt=DRAW_LINEXY: Network of lines is drawn connecting points at which function is defined.

plmeshc: Magnitude colored plot surface mesh with contour.

```
plmeshc (x, y, z, nx, ny, opt, clevel, nlevel);
```

Identical to plmesh but with extra functionalities: the surface mesh can be colored accordingly to the current z value being plotted, a contour plot can be drawn at the base XY plane, and a curtain can be drawn between the plotted function border and the base XY plane.

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. To specify more than one option just add the options, e.g. DRAW_LINEXY + MAG_COLOR

opt=DRAW_LINEX: Lines are drawn showing *z* as a function of *x* for each value of *y*[j].

opt=DRAW_LINEY: Lines are drawn showing *z* as a function of *y* for each value of *x*[i].

opt=DRAW_LINEXY: Network of lines is drawn connecting points at which function is defined.

opt=MAG_COLOR: Each line in the mesh is colored according to the z value being plotted. The color is used from the current colormap 1.

opt=BASE_CONT: A contour plot is drawn at the base XY plane using parameters *nlevel* and *clevel*.

opt=DRAW_SIDES: draws a curtain between the base XY plane and the borders of the plotted function.

clevel (PLFLT *, input)

Pointer to the array that defines the contour level spacing. evaluated.

nlevel (PLINT, input)

Number of elements in the clevel array.

plmkstrm: Creates a new stream and makes it the default

```
plmkstrm (p_strm);
```

Creates a new stream and makes it the default. Differs from using plsstrm, in that a free stream number is found, and returned. Unfortunately, I *have* to start at stream 1 and work upward, since stream 0 is preallocated. One of the *big* flaws in the PLplot API is that no initial, library-opening call is required. So stream 0 must be preallocated, and there is no simple way of determining whether it is already in use or not.

p_strm (PLINT *, output)

Pointer to stream number of the created stream.

plmtex: Write text relative to viewport boundaries

```
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 (const 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, text written parallel to edge.

bv: Bottom of viewport, text written at right angles to edge.

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, text written parallel to edge.

tv: Top of viewport, text written at right angles to edge.

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 (const char *, input)

The string to be written out.

plot3d: Plot 3-d surface plot

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

Plots a three dimensional surface plot within the environment set up by plw3d. The surface is defined by the two-dimensional array *z*[*nx*][*ny*], 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 the Section called *Three Dimensional Surface Plots* in Chapter 3. The only difference between plmesh() and plot3d() is that plmesh() draws the botton side of the surface, while plot3d() only draws the surface as view from the top.

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:

opt=DRAW_LINEX: Lines are drawn showing *z* as a function of *x* for each value of *y*[j].

opt=DRAW_LINEY: Lines are drawn showing *z* as a function of *y* for each value of *x*[i].

opt=DRAW_LINEXY: 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.

plot3dc: Magnitude colored plot surface with contour.

```
plot3dc (x, y, z, nx, ny, opt, clevel, nlevel);
```

Identical to plot3d but with extra functionalities: the surface mesh can be colored accordingly to the current z value being plotted, a contour plot can be drawn at the base XY plane, and a curtain can be drawn between the plotted function border and the base XY plane. The arguments are identical to plmeshc. The only difference between plmeshc() and plot3dc() is that plmeshc() draws the botton side of the surface, while plot3dc() only draws the surface as viewed from the top.

plpage: Begin a new page

```
plpage ();
```

Deprecated. Use the new name, plbop, for this function instead.

plpat: Set area fill pattern

```
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. Specifies the spacing in micrometers between the lines making up the pattern.

plpoin: Plots a character at the specified points

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

Marks a set of *n* points in *x* and *y* using the symbol defined by *code*. If *code* is between 32 and 127, the symbol is simply the corresponding printable ASCII character in the default font.

n (PLINT, input)

Number of points to be marked.

x (PLFLT *, input)

Pointer to array with x coordinates of the points.

y (PLFLT *, input)

Pointer to array with y coordinates of the points.

code (PLINT, input)

Code number for the symbol to be plotted.

plpoin3: Plots a character at the specified points in 3 space

```
plpoin3 (n, x, y, z, code);
```

Marks a set of *n* points in *x*, *y*, and *z* using the symbol defined by *code*. If *code* is between 32 and 127, the symbol is simply the corresponding printable ASCII character in the default font. Setup similar to plline3.

n (PLINT, input)

Number of points to be marked.

x (PLFLT *, input)

Pointer to array with x coordinates of the points.

y (PLFLT *, input)

Pointer to array with y coordinates of the points.

z (PLFLT *, input)

Pointer to array with z coordinates of the points.

code (PLINT, input)

Code number for the symbol to be plotted.

plpoly3: Draw a polygon in 3 space

```
plpoly3 (n, x, y, z, draw, ifcc);
```

Draws a polygon in 3 space defined by *n* points in *x*, *y*, and *z*. Setup like plline3, but differs from that function in that plpoly3 attempts to determine if the polygon is viewable depending on the order of the points within the arrays and the value of *ifcc*. If the back of polygon is facing the viewer, then it isn't drawn. If this isn't what you want, then use plline3 instead.

The points are assumed to be in a plane, and the directionality of the plane is determined from the first three points. Additional points do not *have* to lie on the plane defined by the first three, but if they do not, then the determination of visibility obviously can't be 100% accurate... So if you're 3 space polygons are too far from planar, consider breaking them into smaller polygons. "3 points define a plane" :-).

Bugs: If one of the first two segments is of zero length, or if they are colinear, the calculation of visibility has a 50/50 chance of being correct. Avoid such situations :-). See x18c.c for an example of this problem. (Search for "20.1").

n (PLINT, input)

Number of points defining line.

x (PLFLT *, input)

Pointer to array with x coordinates of points.

y (PLFLT *, input)

Pointer to array with y coordinates of points.

z (PLFLT *, input)

Pointer to array with z coordinates of points.

draw (PLINT *, input)

Pointer to array which controls drawing the segments of the polygon. If *draw[i]* is true, then the polygon segment from index *[i]* to *[i+1]* is drawn, otherwise, not.

ifcc (PLINT, input)

If *ifcc*=1 the directionality of the polygon is determined by assuming the points are laid out in a counter-clockwise order. If *ifcc*=0 the directionality of the polygon is determined by assuming the points are laid out in a clockwise order.

plprec: Set precision in numeric labels

```
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: Select area fill pattern

```
plpsty (n);
```

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

n (PLINT, input)

The desired pattern.

plptex: Write text inside the viewport

```
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 (const char *, input)

The string to be written out.

plreplot: Replays contents of plot buffer to current device/file

```
plreplot ();
```

Replays contents of plot buffer to current device/file.

plrgb: Set line color by red, green

```
plrgb (r, g, b);
```

Set line color by red, green, blue from 0. to 1. Do *not* use this. Only retained for backward compatibility

r (PLFLT, input)

NEEDS DOCUMENTATION

g (PLFLT, input)

NEEDS DOCUMENTATION

b (PLFLT, input)

NEEDS DOCUMENTATION

plrgb1: Set line color by 8-bit RGB values

```
plrgb1 (r, g, b);
```

Set line color by 8-bit RGB values. Do *not* use this. Only retained for backward compatibility.

r (PLINT, input)

NEEDS DOCUMENTATION

g (PLINT, input)

NEEDS DOCUMENTATION

b (PLINT, input)

NEEDS DOCUMENTATION

plschr: Set character size

```
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.

plscmap0: Set color map0 colors by 8-bit RGB values

```
plscmap0 (r, g, b, ncol0);
```

Set color map0 colors using 8-bit RGB values (see the Section called *Color Map0* in Chapter 3). This sets the entire color map – only as many colors as specified will be allocated.

r (PLINT *, input)

Pointer to array with set of unsigned 8-bit integers (0-255) representing the degree of red in the color.

g (PLINT *, input)

Pointer to array with set of unsigned 8-bit integers (0-255) representing the degree of green in the color.

b (PLINT *, input)

Pointer to array with set of unsigned 8-bit integers (0-255) representing the degree of blue in the color.

ncol0 (PLINT, input)

Number of items in the *r*, *g*, and *b* arrays.

plscmap0n: Set number of colors in color map0

```
plscmap0n (ncol0);
```

Set number of colors in color map0 (see the Section called *Color Map0* in Chapter 3). Allocate (or reallocate) color map0, and fill with default values for those colors not previously allocated. The first 16 default colors are given in the `plcol0` documentation. For larger indices the default color is red.

The drivers are not guaranteed to support more than 16 colors.

ncol0 (PLINT, input)

Number of colors that will be allocated in the map0 palette. If this number is zero or less, then the value from the previous call to `plscmap0n` is used and if there is no previous call, then a default value is used.

plscmap1: Set color map1 colors using 8-bit RGB values

```
plscmap1 (r, g, b, ncol1);
```

Set color map1 colors using 8-bit RGB values (see the Section called *Color Map1* in Chapter 3). This also sets the number of colors.

r (PLINT *, input)

Pointer to array with set of unsigned 8-bit integers (0-255) representing the degree of red in the color.

g (PLINT *, input)

Pointer to array with set of unsigned 8-bit integers (0-255) representing the degree of green in the color.

b (PLINT *, input)

Pointer to array with set of unsigned 8-bit integers (0-255) representing the degree of blue in the color.

ncol1 (PLINT, input)

Number of items in the *r*, *g*, and *b* arrays.

plscmap1l: Set color map1 colors using a piece-wise linear relationship

```
plscmap1l (itype, npts, pos, coord1, coord2, coord3, rev);
```

Set color map1 colors using a piece-wise linear relationship between position in the color map (from 0 to 1) and position in HLS or RGB color space (see the Section called *Color Map1* in Chapter 3). May be called at any time.

The idea here is to specify a number of control points that define the mapping between palette 1 input positions (intensities) and HLS (or RGB). Between these points, linear interpolation is used which gives a smooth variation of color with input position. Any number of control points may be specified, located at arbitrary positions, although typically 2 - 4 are enough. Another way of stating this is that we are traversing a given number of lines through HLS (or RGB) space as we move through color map1 entries. The control points at the minimum and maximum position (0 and 1) must always be specified. By adding more control points you can get more variation. One good technique for plotting functions that vary about some expected average is to use an additional 2 control points in the center (position =

0.5) that are the same lightness as the background (typically white for paper output, black for crt), and same hue as the boundary control points. This allows the highs and lows to be very easily distinguished.

Each control point must specify the position in color map1 as well as three coordinates in HLS or RGB space. The first point *must* correspond to position = 0, and the last to position = 1.

The hue is interpolated around the front of the color wheel (red - green - blue - red) unless the rev flag is set, in which case interpolation (between the `i` and `i + 1` control point for `rev[i]`) proceeds around the back (reverse) side. Specifying rev=NULL is equivalent to setting rev[]=0 for every control point.

Table 15-1. Bounds on coordinates

RGB	R	[0, 1]	magnitude
RGB	G	[0, 1]	magnitude
RGB	B	[0, 1]	magnitude
HLS	hue	[0, 360]	degrees
HLS	lightness	[0, 1]	magnitude
HLS	saturation	[0, 1]	magnitude

itype (`PLINT`, input)

0: HLS, 1: RGB

npts (`PLINT`, input)

number of control points

pos (`PLFLT *`, input)

position for each control point

coord1 (`PLFLT *`, input)

first coordinate (H or R) for each control point

coord2 (`PLFLT *`, input)

second coordinate (L or G) for each control point

coord3 (`PLFLT *`, input)

third coordinate (S or B) for each control point

rev (`PLINT *`, input)

reverse flag for each control point (`rev[i]` refers to the interpolation interval between the `i` and `i + 1` control points).

plscmap1n: Set number of colors in color map1

```
plscmap1n (ncol1);
```

Set number of colors in color map1, (re-)allocate color map1, and set default values if this is the first allocation (see the Section called *Color Map1* in Chapter 3).

ncol1 (PLINT, input)

Number of colors that will be allocated in the map1 palette. If this number is zero or less, then the value from the previous call to plscmap1n is used and if there is no previous call, then a default value is used.

plscol0: Set a given color from color map0 by 8 bit RGB value

```
plscol0 (icol0, r, g, b);
```

Set a given color by 8-bit RGB value for color map0 (see the Section called *Color Map0* in Chapter 3). Overwrites the previous (default?) color value for the given index and, thus, does not result in any additional allocation of space for colors.

icol0 (PLINT, input)

Color index. Must be less than the maximum number of colors (which is set by default, by plscmap0n, or even by plscmap0).

r (PLINT, input)

Unsigned 8-bit integer (0-255) representing the degree of red in the color.

g (PLINT, input)

Unsigned 8-bit integer (0-255) representing the degree of green in the color.

b (PLINT, input)

Unsigned 8-bit integer (0-255) representing the degree of blue in the color.

plscolbg: Set the background color (cmap0[0]) by 8-bit RGB value

```
plscolbg (r, g, b);
```

Set the background color (cmap0[0]) by 8-bit RGB value (see the Section called *Color Map0* in Chapter 3).

r (PLINT, input)

Unsigned 8-bit integer (0-255) representing the degree of red in the color.

g (PLINT, input)

Unsigned 8-bit integer (0-255) representing the degree of green in the color.

b (PLINT, input)

Unsigned 8-bit integer (0-255) representing the degree of blue in the color.

plscolor: Used to globally turn color output on/off

```
plscolor (color);
```

Used to globally turn color output on/off for those drivers/devices that support it.

color (PLINT, input)

Color flag (Boolean). If zero, color is turned off. If non-zero, color is turned on.

plscompression: Set device-compression level

```
plscompression (compression);
```

Set device-compression level. Only used for drivers that provide compression. This function, if used, should be invoked before a call to plinit.

compression (PLINT, input)

The desired compression level. This is a device-dependent value.

plsdev: Set the device (keyword) name

```
plsdev (devname);
```

Set the device (keyword) name.

devname (const char *, output)

Pointer to device (keyword) name string.

plsdidev: Set parameters that define current device-space window

```
plsdidev (mar, aspect, jx, jy);
```

Set relative margin width, aspect ratio, and relative justification that define current device-space window. If you want to just use the previous value for any of these, just pass in the magic value PL NOTSET. It is unlikely that one should ever need to change the aspect ratio but it's in there for completeness. If plsdidev is not called the default values of *mar*, *aspect*, *jx*, and *jy* are all 0.

mar (PLFLT, input)

Relative margin width.

aspect (PLFLT, input)

Aspect ratio.

jx (PLFLT, input)

Relative justification in x.

jy (PLFLT, input)

Relative justification in y.

plsdimap: Set up transformation from metafile coordinates

```
plsdimap (dimxmin , dimxmax , dimymin , dimymax , dimxpmm , dimypmm );
```

Set up transformation from metafile coordinates. The size of the plot is scaled so as to preserve aspect ratio. This isn't intended to be a general-purpose facility just yet (not sure why the user would need it, for one).

dimxmin (PLINT, input)

NEEDS DOCUMENTATION

dimxmax (PLINT, input)

NEEDS DOCUMENTATION

dimymin (PLINT, input)

NEEDS DOCUMENTATION

dimymax (PLINT, input)

NEEDS DOCUMENTATION

dimxpmm (PLFLT, input)

NEEDS DOCUMENTATION

dimypmm (PLFLT, input)

NEEDS DOCUMENTATION

plsdiori: Set plot orientation

```
plsdiori (rot);
```

Set plot orientation parameter which is multiplied by 90 to obtain the angle of rotation. Note, arbitrary rotation parameters such as 0.2 (corresponding to 18) are possible, but the usual values for the rotation parameter are 0., 1., 2., and 3. corresponding to 0 (landscape mode), 90 (portrait mode), 180 (seascape mode), and 270 (upside-down mode). If plsdiori is not called the default value of *rot* is 0.

rot (PLFLT, input)

Plot orientation parameter.

plsdiplt: Set parameters that define current plot-space window

```
plsdiplt (xmin , ymin , xmax , ymax );
```

Set relative minima and maxima that define current plot-space window. If plsdiplt is not called the default values of *xmin* , *ymin* , *xmax* , and *ymax* are 0., 0., 1., and 1.

xmin (PLFLT, input)

Relative minimum in x.

ymin (PLFLT, input)

Relative minimum in y.

xmax (PLFLT, input)

Relative maximum in x.

ymax (PLFLT, input)

Relative maximum in y.

plsdiplz: Set parameters incrementally (zoom mode) that define current plot-space window

```
plsdiplz (xmin, ymin, xmax, ymax);
```

Set relative minima and maxima incrementally (zoom mode) that define the current plot-space window. This function has the same effect as plsdiplt if that function has not been previously called. Otherwise, this function implements zoom mode using the transformation `min_used = old_min + old_length*min` and `max_used = old_min + old_length*max` for each axis. For example, if `min = 0.05` and `max = 0.95` for each axis, repeated calls to plsdiplz will zoom in by 10 per cent for each call.

xmin (PLFLT, input)

Relative (incremental) minimum in x.

ymin (PLFLT, input)

Relative (incremental) minimum in y.

xmax (PLFLT, input)

Relative (incremental) maximum in x.

ymax (PLFLT, input)

Relative (incremental) maximum in y.

plsesc: Set the escape character for text strings

```
plsesc (esc);
```

Set the escape character for text strings. From C (in contrast to fortran, see plsescfortran) you pass *esc* as a character. Only selected characters are allowed to prevent the user from shooting himself in the foot (For example, a " " isn't allowed since it conflicts with C's use of backslash as a character escape). Here are the allowed escape characters and their corresponding decimal ASCII values:

"!", ASCII 33

"#", ASCII 35

"$", ASCII 36

"%", ASCII 37

"&", ASCII 38

"*", ASCII 42

"@", ASCII 64

" ", ASCII 94

" ", ASCII 126

esc (char, input)

Escape character.

plsetopt: Set any command-line option

```
plsetopt (opt, optarg);
```

Set any command-line option internally from a program before it invokes plinit. opt is the name of the command-line option and optarg is the corresponding command-line option argument.

opt (const char *, output)

Pointer to string containing the command-line option.

optarg (const char *, output)

Pointer to string containing the argument of the command-line option.

plsfam: Set family file parameters

```
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 the Section called *Family File Output* in Chapter 3 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.

plsfnam: Set output file name

```
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 (`const char *`, input)

Pointer to file name string.

plshades: Shade regions on the basis of value

```
plshades (a, nx, ny, defined, xmin, xmax, ymin, ymax, clevel, nlevel, fill_width,
cont_color, cont_width, fill, rectangular, pltr, pltr_data);
```

Shade regions on the basis of value. This is the high-level routine for making continuous color shaded plots with cmap1 while `plshade` (or `plshade1`) are used for individual shaded regions using either cmap0 or cmap1. `examples/c/x16c.c` shows a number of examples for using this function. See the following discussion of the arguments and the Section called *Contour and Shade Plots* in Chapter 3 for more information.

a (`PLFLT **`, input)

Contains ** pointer to array to be plotted. The array must have been declared as PLFLT a[nx][ny].

nx (`PLINT`, input)

First dimension of array a .

ny (`PLINT`, input)

Second dimension of array a .

defined (`PLINT (*) (PLFLT, PLFLT)`, input)

User function specifying regions excluded from the shading plot. This function accepts x and y coordinates as input arguments and must return 0 if the point is in the excluded region or 1 otherwise. This argument can be NULL if all the values are valid.

xmin (`PLFLT`, input)

Defines the grid coordinates. The data a[0][0] has a position of (xmin,ymin).

xmax (`PLFLT`, input)

Defines the grid coordinates. The data a[0][0] has a position of (xmin,ymin).

ymin (`PLFLT`, input)

Defines the grid coordinates. The data a[0][0] has a position of (xmin,ymin).

ymax (`PLFLT`, input)

Defines the grid coordinates. The data a[0][0] has a position of (xmin,ymin).

clevel (PLFLT *, input)

Pointer to array containing the data levels corresponding to the edges of each shaded region that will be plotted by this function. To work properly the levels should be monotonic.

nlevel (PLINT, input)

Number of shades plus 1 (i.e., the number of shade edge values in *clevel*).

fill_width (PLINT, input)

Defines width used by the fill pattern.

cont_color (PLINT, input)

Defines pen color used for contours defining edges of shaded regions. The pen color is only temporary set for the contour drawing. Set this value to zero or less if no shade edge contours are wanted.

cont_width (PLINT, input)

Defines pen width used for contours defining edges of shaded regions. This value may not be honored by all drivers. The pen width is only temporary set for the contour drawing. Set this value to zero or less if no shade edge contours are wanted.

fill (void (*) (PLINT, PLFLT *, PLFLT *), input)

Routine used to fill the region. Use plfill. Future version of plplot may have other fill routines.

rectangular (PLINT, input)

Set *rectangular* to 1 if rectangles map to rectangles after coordinate transformation with *pltrl*. Otherwise, set *rectangular* to 0. If *rectangular* is set to 1, plshade tries to save time by filling large rectangles. This optimization fails if the coordinate transformation distorts the shape of rectangles. For example a plot in polor coordinates has to have rectangular set to 0.

pltr (void (*) (PLFLT, PLFLT, PLFLT *, PLFLT *, PLPointer) , input)

Pointer to function that defines transformation between indices in array *z* and the world coordinates (C only). Transformation functions are provided in the PLplot library: pltr0 for identity mapping, and pltr1 and pltr2 for arbitrary mappings respectively defined by one- and two-dimensional arrays. In addition, user-supplied routines for the transformation can be used as well. Examples of all of these approaches are given in the Section called *Contour Plots from C* in Chapter 3. The transformation function should have the form given by any of pltr0, pltr1, or pltr2.

pltr_data (PLPointer, input)

Extra parameter to help pass information to pltr0, pltr1, pltr2, or whatever routine that is externally supplied.

plshade: Shade individual region on the basis of value

```
plshade (a, nx, ny, defined, xmin, xmax, ymin, ymax, shade_min, shade_max, sh_cmap,
sh_color, sh_width, min_color, min_width, max_color, max_width, fill, rectangular, pltr,
pltr_data);
```

Shade individual region on the basis of value. Use plshades if you want to shade a number of regions using continuous colors. plshade is identical to plshade1 except for the type of the first parameter. See plshade1 for further discussion.

a (PLFLT **, input)

nx (PLINT, input)

ny (PLINT, input)

defined (PLINT (*) (PLFLT, PLFLT), input)

xmin (PLFLT, input)

xmax (PLFLT, input)

ymin (PLFLT, input)

ymax (PLFLT, input)

shade_min (PLFLT, input)

shade_max (PLFLT, input)

sh_cmap (PLINT, input)

sh_color (PLFLT, input)

sh_width (PLINT, input)

min_color (PLINT, input)

min_width (PLINT, input)

max_color (PLINT, input)

max_width (PLINT, input)

fill (void (*) (PLINT, PLFLT *, PLFLT *), input)

rectangular (PLINT, input)

pltr (void (*) (PLFLT, PLFLT, PLFLT *, PLFLT *, PLPointer) , input)

pltr_data (PLPointer, input)

plshade1: Shade individual region on the basis of value

```
plshade1 (a, nx, ny, defined, xmin, xmax, ymin, ymax, shade_min, shade_max, sh_cmap,
sh_color, sh_width, min_color, min_width, max_color, max_width, fill, rectangular, pltr,
pltr_data);
```

Shade individual region on the basis of value. Use plshades if you want to shade a number of contiguous regions using continuous colors. In particular the edge contours are treated properly in plshades. If you attempt to do contiguous regions with plshade1 (or plshade) the contours at the edge of the shade are partially obliterated by subsequent plots of contiguous shaded regions. plshade1 differs from plshade by the type of the first argument. Look at the argument list below, plcont and the Section called *Contour and Shade Plots* in Chapter 3 for more information about the transformation from grid to world coordinates. Shading NEEDS DOCUMENTATION, but as a stopgap look at how plshade is used in examples/c/x15c.c

a (PLFLT *, input)

Contains array to be plotted. The array must have been declared as PLFLT a[nx][ny].

nx (PLINT, input)

First dimension of array a .

ny (PLINT, input)

Second dimension of array a .

defined (PLINT (*) (PLFLT, PLFLT), input)

User function specifying regions excluded from the shading plot. This function accepts x and y coordinates as input arguments and must return 0 if the point is in the excluded region or 1 otherwise. This argument can be NULL if all the values are valid.

xmin (PLFLT, input)

Defines the grid coordinates. The data a[0][0] has a position of (xmin,ymin).

xmax (PLFLT, input)

Defines the grid coordinates. The data a[0][0] has a position of (xmin,ymin).

ymin (PLFLT, input)

Defines the grid coordinates. The data a[0][0] has a position of (xmin,ymin).

ymax (PLFLT, input)

Defines the grid coordinates. The data a[0][0] has a position of (xmin,ymin).

shade_min (PLFLT, input)

Defines the interval to be shaded. If shade_max ≤ shade_min, plshade1 does nothing.

shade_max (PLFLT, input)

Defines the interval to be shaded. If shade_max ≤ shade_min, plshade1 does nothing.

sh_cmap (PLINT, input)

Defines color map.

sh_color (PLFLT, input)

Defines color map index if cmap0 or color map input value (ranging from 0. to 1.) if cmap1.

sh_width (PLINT, input)

Defines width used by the fill pattern.

min_color (PLINT, input)

Defines pen color, width used by the boundary of shaded region. The min values are used for the shade_min boundary, and the max values are used on the shade_max boundary. Set color and width to zero for no plotted boundaries.

min_width (PLINT, input)

Defines pen color, width used by the boundary of shaded region. The min values are used for the shade_min boundary, and the max values are used on the shade_max boundary. Set color and width to zero for no plotted boundaries.

max_color (PLINT, input)

Defines pen color, width used by the boundary of shaded region. The min values are used for the shade_min boundary, and the max values are used on the shade_max boundary. Set color and width to zero for no plotted boundaries.

max_width (PLINT, input)

Defines pen color, width used by the boundary of shaded region. The min values are used for the shade_min boundary, and the max values are used on the shade_max boundary. Set color and width to zero for no plotted boundaries.

fill (void (*) (PLINT, PLFLT *, PLFLT *), input)

Routine used to fill the region. Use plfill. Future version of plplot may have other fill routines.

rectangular (PLINT, input)

Set *rectangular* to 1 if rectangles map to rectangles after coordinate transformation with *pltrl*. Otherwise, set *rectangular* to 0. If *rectangular* is set to 1, plshade tries to save time by filling large rectangles. This optimization fails if the coordinate transformation distorts the shape of rectangles. For example a plot in polor coordinates has to have rectangular set to 0.

pltr (void (*) (PLFLT, PLFLT, PLFLT *, PLFLT *, PLPointer) , input)

Pointer to function that defines transformation between indices in array *z* and the world coordinates (C only). Transformation functions are provided in the PLplot library: pltr0 for identity mapping, and pltr1 and pltr2 for arbitrary mappings respectively defined by one- and two-dimensional arrays. In addition, user-supplied routines for the transformation can be used as well. Examples of all of these approaches are given in the Section called *Contour Plots from C* in Chapter 3. The transformation function should have the form given by any of pltr0, pltr1, or pltr2.

pltr_data (PLPointer, input)

Extra parameter to help pass information to pltr0, pltr1, pltr2, or whatever routine that is externally supplied.

plsmaj: Set length of major ticks

```
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.

plsmem: Set the memory area to be plotted

```
plsmem (maxx, maxy, plotmem);
```

Set the memory area to be plotted (with the "mem" driver) as the dev member of the stream structure. Also set the number of pixels in the memory passed in *plotmem*, which is a block of memory *maxy* by *maxx* by 3 bytes long, say: 480 x 640 x 3 (Y, X, RGB)

This memory will be freed by the user!

maxx (PLINT, input)

Size of memory area in the X coordinate.

maxy (PLINT, input)

Size of memory area in the Y coordinate.

plotmem (void *, input)

Pointer to the beginning of the user-supplied memory area.

plsmin: Set length of minor ticks

```
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: Set orientation

```
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: Set page parameters

```
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 (PLFLT, input)

Number of pixels, x.

yp (PLFLT, input)

Number of pixels, y.

xleng (PLINT , input)

Page length, x.

yleng (PLINT, input)

Page length, y.

xoff (PLINT, input)

Page offset, x.

yoff (PLINT, input)

Page offset, y.

plspause: Set the pause (on end-of-page) status

```
plspause (pause);
```

Set the pause (on end-of-page) status.

pause (PLINT, input)

If *pause* = 1 there will be a pause on end-of-page for those drivers which support this. Otherwise not.

plsstrm: Set current output stream

```
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, input)

The current stream number.

plssub: Set the number of subwindows in x and y

```
plssub (nx, ny);
```

Set the number of subwindows in x and y.

nx (PLINT, input)

Number of windows in x direction (i.e., number of window columns).

ny (PLINT, input)

Number of windows in y direction (i.e., number of window rows).

plssym: Set symbol size

```
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: Initialization

```
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: Initialization

```
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 (const 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.

plstripa: Add a point to a stripchart

```
plstripa (id, p, x, y);
```

Add a point to a given pen of a given stripchart. There is no need for all pens to have the same number of points or to be equally sampled in the x coordinate. Allocates memory and rescales as necessary.

id (PLINT, input)

Identification number (set up in plstripc) of the stripchart.

p (PLINT, input)

Pen number (ranges from 0 to 3).

x (PLFLT, input)

X coordinate of point to plot.

y (PLFLT, input)

Y coordinate of point to plot.

plstripc: Create a 4-pen stripchart

```
plstripc (id, xspec, yspec, xmin, xmax, xjump, ymin, ymax, xlpos, ylpos, y_ascl, acc,
colbox, collab, colline, styline, legline[], labx, laby, labtop);
```

Create a 4-pen stripchart, to be used afterwards by plstripa

id (PLINT *, output)

Identification number of stripchart to use on plstripa and plstripd.

xspec (char *, input)

X-axis specification as in plbox.

yspec (char *, input)

Y-axis specification as in plbox.

xmin (PLFLT, input)

Initial coordinates of plot box; they will change as data are added.

xmax (PLFLT, input)

Initial coordinates of plot box; they will change as data are added.

xjump (PLFLT, input)

When x attains *xmax*, the length of the plot is multiplied by the factor (1 + *xjump*).

ymin (PLFLT, input)

Initial coordinates of plot box; they will change as data are added.

ymax (PLFLT, input)

Initial coordinates of plot box; they will change as data are added.

xlpos (PLFLT, input)

X legend box position (range from 0 to 1).

ylpos (PLFLT, input)

Y legend box position (range from 0 to 1).

y_ascl (PLINT, input)

Autoscale y between x jumps (1) or not (0).

acc (PLINT, input)

Accumulate strip plot (1) or slide (0).

colbox (PLINT, input)

Plot box color index (cmap0).

collab (PLINT, input)

Legend color index (cmap0).

colline (PLINT *, input)

Pointer to array with color indices (cmap0) for the 4 pens.

styline (PLINT *, input)

Pointer to array with line styles for the 4 pens.

legline (char **, input)

Pointer to character array containing legends for the 4 pens.

labx (char *, input)

X-axis label.

laby (char *, input)

Y-axis label.

labtop (char *, input)

Plot title.

plstripd: Deletes and releases memory used by a stripchart

```
plstripd (id);
```

Deletes and releases memory used by a stripchart.

id (PLINT, input)

Identitification number of stripchart to delete.

plstyl: Set line style

```
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.(see also 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.

plsurf3d: Plot shaded 3-d surface plot

```
plsurf3d (x, y, z, nx, ny, opt, clevel, nlevel);
```

Plots a three dimensional shaded surface plot within the environment set up by plw3d. The surface is defined by the two-dimensional array *z*[*nx*][*ny*], 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. For further details see the Section called *Three Dimensional Surface Plots* in Chapter 3.

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. To specify more than one option just add the options, e.g. FACETED + SURF CONT

opt=FACETED: Network of lines is drawn connecting points at which function is defined.

opt=BASE_CONT: A contour plot is drawn at the base XY plane using parameters *nlevel* and *clevel*.

opt=SURF_CONT: A contour plot is drawn at the surface plane using parameters *nlevel* and *clevel*.

opt=DRAW_SIDES: draws a curtain between the base XY plane and the borders of the plotted function.

clevel (PLFLT *, input)

Pointer to the array that defines the contour level spacing. evaluated.

nlevel (PLINT, input)

Number of elements in the clevel array.

plsvpa: Specify viewport in absolute coordinates

```
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 bottom edge of the subpage in millimeters.

plsxax: Set x axis parameters

```
plsxax (digmax, digits);
```

Sets values of the *digmax* and *digits* flags for the x axis. See the Section called *Annotating the Viewport* in Chapter 3 for more information.

digmax (PLINT, input)

Variable to set the maximum number of digits for the x axis. 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 either of these functions by calling plgxax.

plsyax: Set y axis parameters

```
plsyax (digmax, digits);
```

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

digmax (PLINT, input)

Variable to set the maximum number of digits for the y axis. 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 either of these functions by calling plgyax.

plsym: Plots a symbol at the specified points

```
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: Set z axis parameters

```
plszax (digmax, digits);
```

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

digmax (PLINT, input)

Variable to set the maximum number of digits for the z axis. 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 either of these functions by calling plgzax.

pltext: Switch to text screen

```
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 (see also plgra).

plvasp: Specify viewport using aspect ratio only

```
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: Specify viewport using coordinates and aspect ratio

```
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: Specify viewport using coordinates

```
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: Select standard 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: Set up window for 3-d plotting

```
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 the Section called *Three Dimensional Surface Plots* in Chapter 3.

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: Set pen width

```
plwid (width);
```

Sets the pen width.

width (PLINT, input)

The desired pen width. If *width* is negative or the same as the previous value no action is taken. *width* = 0 should be interpreted as as the minimum valid pen width for the device. The interpretation of positive *width* values is also device dependent.

plwind: Specify world coordinates of viewport boundaries

```
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.

plxormod: Enter or leave xor mode

```
plxormod (mode, status);
```

Enter (mode !=0) or leave (mode == 0) xor mode for those drivers (e.g., the xwin driver) that support it. Enables erasing plots by drawing twice the same line, symbol, etc. If driver is not capable of xor operation returns status of 0.

mode (PLINT, input)

mode != 0 means enter xor mode and mode == 0 means leave xor mode.

status (PLINT *, output)

Pointer to status. Returned status == 1 (0) means driver is capable (incapable) of xor mode.

Chapter 16. The Specialized C API for PLplot

The purpose of this chapter is to document the API for every C function in PLplot (other than language bindings) that is *not* part of the common API that has already been documented in Chapter 15.

This chapter is a work that is just starting. There are many C functions in the code base that are not part of the common API, and we haven't even gotten to the point of listing them all. What gets documented here now is whatever C-explicit code we are trying to understand at the time.

plP_checkdriverinit: Checks to see if any of the specified drivers have been initialized

```
plP_checkdriverinit (list);
```

Checks to see if any of the specified drivers have been initialized. Function tests a space-delimited list of driver names to see how many of the given drivers have been initialized, and how often. The return code of the function is: `0` if no matching drivers were found to have been initialized; `-1` if an error occurred allocating the internal buffer; or, a positive number indicating the number of streams encountered that belong to drivers on the provided list. This function invokes `plP_getinitdriverlist` internally to get a *complete* list of drivers that have been initialized in order to compare with the driver names specified in the argument list to `plP_checkdriverinit`.

list (`char *`, input)

Pointer to character string specifying a space-delimited list of driver names, e.g., `"bmp jpeg tiff"`.

plP_getinitdriverlist: Get the initialized-driver list

```
plP_getinitdriverlist (text_buffer);
```

Get the initialized-driver list. Function returns a space-delimited list of the currently initialized drivers or streams. If more than one stream is using the same driver, then its name will be returned more than once. The function can be analogously thought of as also returning the names of the active streams. Invoked internally by `plP_checkdriverinit`.

text_buffer (`char *`, output)

Pointer to a user-allocated buffer to hold the result. The user must ensure the buffer is big enough to hold the result.

plabort: Error abort

```
plabort (message);
```

This routine is to be used when something goes wrong that doesn't require calling `plexit` but for which there is no useful recovery. It calls the abort handler defined via `plsabort`, does some cleanup and returns. The user can supply his/her own abort handler and pass it in via `plsabort`.

message (`char *`, input)

Abort message.

plexit: Error exit

```
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 the handler prvided by `plsexit` and then exits. If cleanup needs to be done in the driver program then the user may want to supply his/her own exit handler and pass it in via `plsexit`. This function should either call `plend` before exiting, or simply return.

message (`char *`, input)

Error message.

plgfile: Get output file handle

```
plgfile (file);
```

Gets the current output file handle, if applicable.

file (`FILE **`, output)

File pointer to current output file.

plsabort: Set abort handler

```
plsabort (handler);
```

Sets an optional user abort handler. See `plabort` for details.

handler (`void (*) (char *)`, input)

Error abort handler.

plsexit: Set exit handler

```
plsexit (handler);
```

Sets an optional user exit handler. See `plexit` for details.

handler (int (*) (char *), input)

Error exit handler.

plsfile: Set output file handle

```
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.

pltr0: Identity transformation for grid to world mapping

```
pltr0 (x, y, tx, ty, pltr_data);
```

Identity transformation for grid to world mapping. This routine can be used both for plcont and plshade. See also the Section called *Contour Plots from C* in Chapter 3 and the Section called *Shade Plots from C* in Chapter 3.

x (PLFLT, input)

X-position in grid coordinates.

y (PLFLT, input)

Y-position in grid coordinates.

tx (PLFLT *, output)

X-position in world coordinates.

ty (PLFLT *, output)

Y-position in world coordinates.

pltr_data (PLPointer, input)

Pointer to additional input data that is passed as an argument to plcont or plshade and then on to the grid to world transformation routine.

pltr1: Linear interpolation for grid to world mapping using singly dimensioned coord arrays

```
pltr1 (x, y, tx, ty, pltr_data);
```

Linear interpolation for grid to world mapping using singly dimensioned coord arrays. This routine can be used both for `plcont` and `plshade`. See also the Section called *Contour Plots from C* in Chapter 3 and the Section called *Shade Plots from C* in Chapter 3.

x (`PLFLT`, input)

X-position in grid coordinates.

y (`PLFLT`, input)

Y-position in grid coordinates.

tx (`PLFLT *`, output)

X-position in world coordinates.

ty (`PLFLT *`, output)

Y-position in world coordinates.

pltr_data (`PLPointer`, input)

Pointer to additional input data that is passed as an argument to `plcont` or `plshade` and then on to the grid to world transformation routine.

pltr2: Linear interpolation for grid to world mapping using doubly dimensioned coord arrays (column dominant, as per normal C 2d arrays)

```
pltr2 (x, y, tx, ty, pltr_data);
```

Linear interpolation for grid to world mapping using doubly dimensioned coord arrays (column dominant, as per normal C 2d arrays). This routine can be used both for `plcont` and `plshade`. See also the Section called *Contour Plots from C* in Chapter 3 and the Section called *Shade Plots from C* in Chapter 3.

x (`PLFLT`, input)

X-position in grid coordinates.

y (`PLFLT`, input)

Y-position in grid coordinates.

tx (`PLFLT *`, output)

X-position in world coordinates.

ty (`PLFLT *`, output)

Y-position in world coordinates.

pltr_data (`PLPointer`, input)

Pointer to additional input data that is passed as an argument to `plcont` or `plshade` and then on to the grid to world transformation routine.

Chapter 17. The Specialized Fortran API for PLplot

The purpose of this Chapter is to document the API for each Fortran function in PLplot that differs substantially (usually in argument lists) from the common API that has already been documented in Chapter 15.

Normally, the common API is wrapped in such a way for Fortran that there is and one-to-one correspondence between each fortran and C argument (see Chapter 7 for discussion). However, for certain routines documented in this chapter the fortran argument lists necessarily differ substantially from the C versions.

This chapter is incomplete and NEEDS DOCUMENTATION of, e.g., the fortran equivalent of the plshade C routines.

plcon0: Contour plot, identity mapping for fortran

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

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 the Section called *Contour and Shade Plots* in Chapter 3 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 indices to consider.

ky, *ly* (PLINT, input)

Range of y indices 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: Contour plot, general 1-d mapping for fortran

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

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 the Section called *Contour and Shade Plots* in Chapter 3 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 indices to consider.

ky, ly (PLINT, input)

Range of y indices 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 plcont using the built-in transformation function pltr1 for the same capability.

plcon2: Contour plot, general 2-d mapping for fortran

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

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 the Section called *Contour and Shade Plots* in Chapter 3 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 indices to consider.

ky, ly (PLINT, input)

Range of y indices 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 plcont using the built-in transformation function pltr2 for the same capability.

plcont: Contour plot, fixed linear mapping for fortran

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

When called from Fortran, this routine has the same effect as when invoked from C. The interpretation of all parameters (see plcont) is also the same except there is no transformation function supplied as the last parameter. Instead, a 6-element array specifying coefficients to use in the transformation is supplied via the named common block plplot (see code). Since this approach is somewhat inflexible, the user is recommended to call either of plcon0, plcon1, or plcon2 instead.

plmesh: Plot surface mesh for fortran

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

When called from Fortran, this routine has the same effect as when invoked from C. The interpretation of all parameters (see plmesh) is also the same except there is an additional parameter given by:

mx (PLINT, input)

Length of array in x direction, for plotting subarrays.

plot3d: Plot 3-d surface plot for fortran

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

When called from Fortran, this routine has the same effect as when invoked from C. The interpretation of all parameters (see plot3d) is also the same except there is an additional parameter given by:

mx (PLINT, input)

Length of array in x direction, for plotting subarrays.

plsesc: Set the escape character for text strings for fortran

```
plsesc (esc);
```

Set the escape character for text strings. From Fortran it needs to be the decimal ASCII value. Only selected characters are allowed to prevent the user from shooting himself in the foot (For example, a " " isn't allowed since it conflicts with C's use of backslash as a character escape). Here are the allowed escape characters and their corresponding decimal ASCII values:

"!", ASCII 33

"#", ASCII 35

"$", ASCII 36

"%", ASCII 37

"&", ASCII 38

"*", ASCII 42

"@", ASCII 64

" ", ASCII 94

" ", ASCII 126

esc (char, input)

NEEDS DOCUMENTATION

Chapter 18. Notes for each Operating System that We Support

The purpose of this Chapter is to present notes for each operating system that we support. Although we have some support for a number of operating systems, we only have notes for Linux/Unix systems at this point. NEEDS DOCUMENTATION

Linux/Unix Notes

Linux/Unix Configure, Build, and Installation

Here is the short story:

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

The longer story is there are a lot of possible configure options. Two of the more important configure options are `--prefix` and `--with-double`. Here is the complete list of configuration options:

```
./configure --help
No defaults file found, performing full configure.
Usage: configure [options] [host]
Options: [defaults in brackets after descriptions]
Configuration:
  --cache-file=FILE       cache test results in FILE
  --help                  print this message
  --no-create             do not create output files
  --quiet, --silent       do not print 'checking...' messages
  --version               print the version of autoconf that created configure
Directory and file names:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [/usr/local/plplot]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [same as prefix]
  --bindir=DIR            user executables in DIR [EPREFIX/bin]
  --sbindir=DIR           system admin executables in DIR [EPREFIX/sbin]
  --libexecdir=DIR        program executables in DIR [EPREFIX/libexec]
  --datadir=DIR           read-only architecture-independent data in DIR
                          [PREFIX/share]
  --sysconfdir=DIR        read-only single-machine data in DIR [PREFIX/etc]
  --sharedstatedir=DIR    modifiable architecture-independent data in DIR
                          [PREFIX/com]
  --localstatedir=DIR     modifiable single-machine data in DIR [PREFIX/var]
  --libdir=DIR            object code libraries in DIR [EPREFIX/lib]
  --includedir=DIR        C header files in DIR [PREFIX/include]
  --oldincludedir=DIR     C header files for non-gcc in DIR [/usr/include]
  --infodir=DIR           info documentation in DIR [PREFIX/info]
  --mandir=DIR            man documentation in DIR [PREFIX/man]
  --srcdir=DIR            find the sources in DIR [configure dir or ..]
  --program-prefix=PREFIX prepend PREFIX to installed program names
  --program-suffix=SUFFIX append SUFFIX to installed program names
  --program-transform-name=PROGRAM
```

```
                        run sed PROGRAM on installed program names
Host type:
  --build=BUILD         configure for building on BUILD [BUILD=HOST]
  --host=HOST           configure for HOST [guessed]
  --target=TARGET       configure for TARGET [TARGET=HOST]
Features and packages:
  --disable-FEATURE     do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE     do not use PACKAGE (same as --with-PACKAGE=no)
  --x-includes=DIR      X include files are in DIR
  --x-libraries=DIR     X library files are in DIR
--enable and --with options recognized:
  --with-defaults       source defaults file at startup (yes)
  --with-debug          compile with debugging (no)
  --with-opt            compile with optimization (yes)
  --with-double         use double precision floats (no)
  --with-profile        turn on profiling option (no)
  --with-shlib          build shared libraries (yes)
  --with-gcc            use gcc to compile C and C++ code (yes)
  --with-warn           enable all compilation warnings (no)
  --with-dbmalloc       link with libdbmalloc (no)
  --with-pkgdir=DIR     locate libraries and includes under DIR
  --with-fseek          use fseek/ftell rather than fsetpos/fgetpos (no)
  --with-rpath          link libraries with -rpath option (yes)
  --enable-f77          compile Fortran-77 interface code (yes)
  --enable-cxx          compile C++ interface code (yes)
  --enable-python       compile python interface code (yes)
  --enable-tcl          compile Tcl interface code (yes)
  --enable-itcl         enable incr Tcl interface code (yes)
  --with-x              use the X Window System
  --with-gtk-prefix=PFX   Prefix where GTK is installed (optional)
  --with-gtk-exec-prefix=PFX Exec prefix where GTK is installed (optional)
  --disable-gtktest     Do not try to compile and run a test GTK program
  --with-gnome-includes   Specify location of GNOME headers
  --with-gnome-libs     Specify location of GNOME libs
  --with-gnome          Specify prefix for GNOME files
  --enable-plmeta       enable plmeta device driver ()
  --enable-null         enable null device driver ()
  --enable-xterm        enable xterm device driver ()
  --enable-tek4010      enable tek4010 device driver ()
  --enable-tek4107      enable tek4107 device driver ()
  --enable-mskermit     enable mskermit device driver ()
  --enable-conex        enable conex device driver ()
  --enable-linuxvga     enable linuxvga device driver ()
  --enable-vlt          enable vlt device driver ()
  --enable-versaterm    enable versaterm device driver ()
  --enable-dg300        enable dg300 device driver ()
  --enable-ps           enable ps device driver ()
  --enable-xfig         enable xfig device driver ()
  --enable-ljii         enable ljii device driver ()
  --enable-hp7470       enable hp7470 device driver ()
  --enable-hp7580       enable hp7580 device driver ()
  --enable-lj_hpgl      enable lj_hpgl device driver ()
  --enable-imp          enable imp device driver ()
  --enable-xwin         enable xwin device driver (yes)
  --enable-tk           enable tk device driver (yes)
```

```
  --enable-pbm            enable pbm device driver ()
  --enable-gnome          enable gnome device driver (no)
```

The configure script looks for default configuration options first in `./cf_plplot.in`. If that file is not found, the script then looks in `$HOME/config/cf_plplot.in`. Finally, if neither file is found or if the found file does not have a particular default option, then the script uses the above defaults. Here is one example of a default configuration file. Adapt this for your needs or else use the command-line parameters for the configuration file.

```
# --*-sh-*-------------------------------------------------------------
#
# PLplot configure script default variables.
#
#
# Note: the internal representation of the --with-<option> and
# --enable-<option> command line switches actually uses an underscore,
# e.g. with_<option> and enable_<option>.  Don't forget!
#
# ---------------------------------------------------------------------

# Method to turn off Fortran and C++ bindings.

enable_cxx="no"
enable_f77="no"

# Devices are selected by --enable or --disable on the command line, but
# only shell variables of the form enable_<option> are recognized here.

enable_tek4010="no"
enable_mskermit="no"
enable_conex="no"
enable_vlt="no"
enable_versaterm="no"
enable_xfig="no"

enable_dg300="no"
enable_imp="no"
enable_tek4107="no"
enable_hp7470="no"
enable_hp7580="no"
```

Linux/Unix Building of C Programmes that Use the Installed PLplot Libraries

This is incomplete. NEEDS DOCUMENTATION. `$prefix/bin/plplot-config` is a useful tool for helping with building of C programmes that use the PLplot libraries.

```
./plplot-config --help
Usage: plplot-config [OPTIONS]
Options:
        [--prefix[=DIR]]
 [--version]
 [--libs]
 [--cflags]
 [--with-c++]
 [--help]
```

For example, the `--cflags` parameter displays the flags for compiling, and the `--libs` parameter displays the flags for linking your application. The displayed flags are exactly consistent with the configuration specified when PLplot was last built and installed.