PGPLOT

GRAPHICS SUBROUTINE LIBRARY

T. J. Pearson

June 1989

California Institute of Technology
Pasadena, California 91125
(818) 395–4980

PGPLOT, Version 4.9, June 1989
(Subroutine descriptions updated to Version 4.9D)

Please address comments on PGPLOT and this manual, including requests for copies, bug reports, and requests for improvements, to the author:

Tim Pearson,
Astronomy Department,
Caltech 105–24,
Pasadena 91125,
USA.

Telephone: +1 818 395-4980
INTERNET: tjp@astro.caltech.edu

Contents

Chapter 1

INTRODUCTION

1.1 PGPLOT

PGPLOT is a Fortran subroutine package for drawing simple scientific graphs on various graphics display devices. It was originally developed for use with astronomical data reduction programs in the Caltech Astronomy department.

This manual is intended for the Fortran programmer who wishes to write a program generating graphical output. For most applications, the program can be device-independent, and the output can be directed to the appropriate device at run time. The output device is described by a "device specification," discussed below. The programmer can build a specific device specification into the program, but it is better to make this a parameter which the user of the program can supply.

All the examples in this manual use standard Fortran-77. PGPLOT itself is written mostly in standard Fortran-77, with a few non-standard, system-dependent subroutines. At Caltech, it runs under the VAX/VMS, Convex-UNIX, and Sun-UNIX operating systems.

1.2 This Manual

This manual is intended both as a tutorial introduction to PGPLOT and as a reference manual. The remainder of this chapter describes some fundamentals: how to include the PGPLOT library in your program, and the types of graphic devices that PGPLOT can use.

Chapter 2 is tutorial: it presents a Fortran program for drawing a graph using the minimum number of PGPLOT subroutines, and explains what each of these subroutines does. After reading this chapter, you should be able to write your own PGPLOT program, although it may be helpful to refer to the individual subroutine descriptions in Appendix A.

The basic features of PGPLOT are introduced in Chapters 3, 4, and 5. Chapter 3 explains the positioning and scaling of plots on the page, Chapter 4 describes the basic ("primitive") routines for drawing lines, writing text, drawing graph markers, and shading areas, and Chapter 5 describes the routines for changing the "attributes" of these primitives: color, line-style, line-width, text font, etc.

Chapter 6 describes some "high level" routines that use the primitive routines to build up more complicated pictures: *e.g.*, function plots, histograms, bar charts, and contour maps.

Chapter 7 describes PGPLOT's capabilities for "interactive" graphics, whereby the user of the PGPLOT program can control its action with a cursor, joystick, mouse, etc.

Chapter 8 describes the use of "metafiles". A metafile is a disk file in which a device-independent representation of a graphics image can be stored. A translation program allows the image to be displayed on any supported device.

There are six appendices. Appendix A is a list of all the PGPLOT routines, with detailed instructions for their use. Appendix B shows the complete set of PGPLOT characters and symbols that can be used for annotating graphs. Appendix C is intended for those who want to install PGPLOT on another machine. Appendix D gives details of the devices supported by PGPLOT. Appendix E provides instructions for programmers who want to extend PGPLOT to support other devices. Appendix F describes how PGPLOT subroutines can be called from a program written in the C language.

1.3 Using PGPLOT

In order to use PGPLOT subroutines, you will need to link your program with the graphics subroutine library.

VAX/VMS On the Caltech Astronomy VAX computers, the graphics subroutine library is scanned automatically by the `LINK` command, so the following sequence of instructions suffices to compile, link, and run a graphics program `EXAMPLE.FOR`:

```
$ FORTRAN EXAMPLE
$ LINK EXAMPLE
$ RUN EXAMPLE
```

On other VMS computers, the automatic search of the graphics library may not occur. You will then need to include the graphics library explicitly by using a `LINK` commands like the following:

```
$ LINK EXAMPLE,PGPLOT_DIR:GRPSHR/LIB
```

The PGPLOT subroutines are not included in your `.EXE` file, but are fetched from a *shareable image* when you execute the `RUN` command. This makes the `.EXE` file much smaller, and means that the program need not be re-linked when changes are made to the graphics subroutines; but the `.EXE` file can only be run on a machine that has a copy of the shareable image and is running a compatible version of VAX/VMS. For more information, see Appendix C.

Unix The following assumes that the PGPLOT library `libpgplot.a` has been installed in a standard location where the loader can find it. To compile, link, and run a graphics program `example.f`:

```
fc -o example example.f -lpgplot
example
```

Unlike the VMS version, the PGPLOT routines are included in the executable file.

1.4 Graphics Devices

Graphics devices fall into two classes: devices which produce a hardcopy output, usually on paper; and interactive devices, which usually display the plot on a TV monitor. Some of the interactive devices allow modification to the displayed picture, and some have a movable cursor which can be used as a graphical input device. There is also a "null device," to which unwanted graphical output can be directed. Hardcopy devices are not used interactively. One must first create a disk file and then send it to the appropriate device with a print or copy command. Consult Appendix D (or your System Manager) to determine the appropriate device-specific command.

A PGPLOT graphical output device is described by a "device specification" that consists of two parts, separated by a slash (/): the *device name* or *file name*, and the *device type*.

Device name The device name or file name is the name by which the output device is known to the operating system. For most hardcopy devices, this should be the name of a disk file, while for interactive devices, it should be the name of a device of the appropriate type; in both cases, the name should be specified according to the syntax of the operating system in use. If the device or file name includes a slash (`/`), enclose the name in double quotation marks (`"`). If the device name is omitted from the device specification, a default device is used, the default depending on the device type (see Appendix D). In Unix, device and file names are case-sensitive.

Device type The device type tells PGPLOT what sort of graphical device it is. Appendix D lists the device types available at the time of writing, together with the names by which they are known to PGPLOT. If the device type is omitted, a system-dependent default type is assumed (this is the value of the "environment variable" `PGPLOT_TYPE`, and on Phobos and Deimos it is "Printronix"). The device type is not case-sensitive: you can use uppercase or lowercase letters, or a mixture of the two.

Examples (VMS)

Tektronix 4006/4010 terminal: `TTA4/TEK` (device `_TTA4:`).

Grinnell image display: `/GRIN`.

Disk file, Printronix format: `SYS$SCRATCH:PLOT.DAT/PRIN`.

Disk file, Versatec format, with the output file on a different DECnet node: `DEIMOS::XPLOT.DAT/PRIN`.

Disk file in default format in default directory: `PGPLOT.LIS`.

Examples (Unix)

Tektronix 4006/4010 terminal: `/TEK` (the logged-in terminal).

IVAS image display: `/IVAS`.

Disk file, Printronix format: `"/scr/tjp/plot.dat"/PRIN`.

Disk file in default format in default directory: `pgplot.lis`.

1.5 Environment variables

The run-time behavior of PGPLOT can be modified by defining one or more *environment variables*. The variables have names which begin with `PGPLOT_`. In VMS, they are logical names; in Unix, they are Unix environment variables.

To set the value of a variable in VMS (DCL):

```
$ DEFINE PGPLOT_ENVOPT VG
```

In Unix (csh):

```
setenv PGPLOT_ENVOPT VG
```

Quotation marks may be required around the value (double-quotes in VMS, single quotes in Unix) to prevent interpretation of special characters by the command interpreter.

To unset a variable in VMS or Unix:

```
$ DEASSIGN PGPLOT_ENVOPT
unsetenv PGPLOT_ENVOPT
```

The following are some of the environment variables currently in use:

- PGPLOT_ENVOPT: this variable provides additional options for the PGENV subroutine (see description in Appendix A).
- PGPLOT_FONT: the name of the binary file containing character font digitization, *e.g.*, PGPLOT_FONT = "/usr/tjp/grfont.dat".
- PGPLOT_IDENT: if this variable is defined (with any value), the user name and time are written at the lower right corner of the plot by routine PGEND (hardcopy devices only), *e.g.*, PGPLOT_IDENT = YES.
- PGPLOT_TYPE: the plot type to be used in PGPLOT when a device specification omits the type, *e.g.*, PGPLOT_TYPE = QMS.
- PGPLOT_BUFFER: controls buffering (see Chapter 7).

Chapter 2

SIMPLE USE OF PGPLOT

2.1 Introduction

This chapter introduces the basic subroutines needed to create a graph using PGPLOT, by way of a concrete example. It does not describe all the capabilities of PGPLOT; these are presented in later chapters.

A graph is composed of several elements: a box or axes delineating the graph and indicating the scale, labels if required, and one or more points or lines. To draw a graph you need to call at least four of the PGPLOT subroutines:

1. `PGBEGIN`, to start up PGPLOT and specify the device you want to plot on;
2. `PGENV`, to define the range and scale of the graph, and draw labels, axes etc;
3. one or more calls to `PGPOINT` or `PGLINE` or both, or other drawing routines, to draw points or lines.
4. `PGEND` to close the plot.

To draw more than graph on the same device, repeat steps (2) and (3). It is only necessary to call `PGBEGIN` and `PGEND` once each, unless you want to plot on more than one device.

This chapter presents a very simple example program to demonstrate the above four steps.

2.2 An Example

A typical application of PGPLOT is to draw a set of measured data points and a theoretical curve for comparison. This chapter describes a simple program for drawing such a plot; in this case there are five data points and the theoretical curve is $y = x^2$. Here is the complete Fortran code for the program:

```
      PROGRAM SIMPLE
      REAL XR(100), YR(100)
      REAL XS(5), YS(5)
      DATA XS/1.,2.,3.,4.,5./
      DATA YS/1.,4.,9.,16.,25./
      CALL PGBEGIN(0,'?',1,1)
      CALL PGENV(0.,10.,0.,20.,0,1)
      CALL PGLABEL('(x)', '(y)', 'A Simple Graph')
      CALL PGPOINT(5,XS,YS,9)
      DO 10 I=1,60
          XR(I) = 0.1*I
          YR(I) = XR(I)**2
   10 CONTINUE
      CALL PGLINE(60,XR,YR)
      CALL PGEND
      END
```

The following sections of this chapter describe how the program works, and the resulting plot is shown in Figure 2.1.

2.3 Data Initialization

We shall store the x and y coordinates of the five data points in arrays XS and YS. For convenience, this program defines the values in DATA statements, but a more realistic program might read them from a file. Arrays XR and YR will be used later in the program for the theoretical curve.

```
      REAL XR(100), YR(100)
      REAL XS(5), YS(5)
      DATA XS/1.,2.,3.,4.,5./
      DATA YS/1.,4.,9.,16.,25./
```

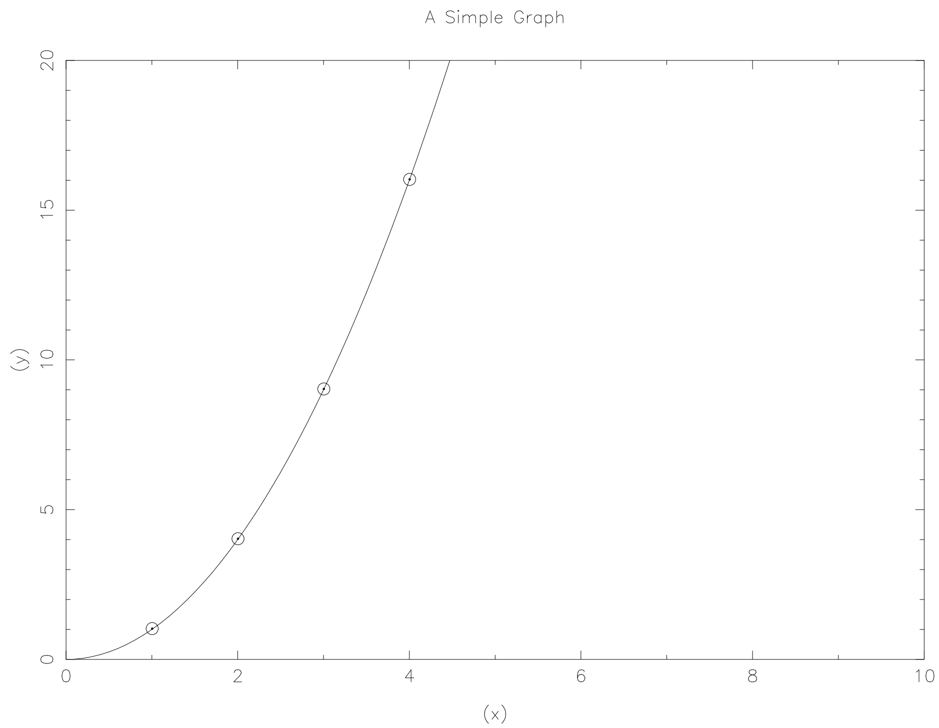

Figure 2.1 Output from example program.

2.4 Starting PGPLOT

The first thing the program must do is to start up PGPLOT and select the graphics device for output:

```
      CALL PGBEGIN(0,'?',1,1)
```

Subroutine `PGBEGIN` has four arguments:

1. The first argument is present for historical reasons. It should always be set to zero (0).
2. The second argument is a character string which gives a "device specification" for the interactive graphics device or disk file for hardcopy graphics (see Chapter 1 and Appendix D). This program makes use of a special shorthand feature of PGPLOT, however: if this argument is set to `'?'`, the program will ask the user to supply the device specification at run-time.

3, 4. The last two arguments are described in §3.2. Usually they are both set to 1, as in this example.

2.5 Defining Plot Scales and Drawing Axes

Subroutine `PGENV` starts a new picture and defines the range of variables and the scale of the plot. `PGENV` also draws and labels the enclosing box and the axes if requested. In this case, the x-axis of the plot will run from 0.0 to 10.0 and the y axis will run from 0.0 to 20.0.

```
CALL PGENV(0.,10.,0.,20.,0,1)
```

`PGENV` has six arguments:

1, 2. the left and right limits for the x (horizontal) axis (real numbers, not integers).

3, 4. the bottom and top limits for the y (vertical) axis (also real numbers).

5. If this (integer) argument is 1, the scales of the x-axis and y-axis (in units per inch) will be equal; otherwise the axes will be scaled independently. In this case we have not requested equal scales.
6. This argument controls whether an enclosing box, tick-marks, numeric labels, and/or a grid will be put on the graph. The recommended value is 0. Some of the allowed values are:
   - -2: no annotation;
   - -1: draw box only;
   - 0: draw box, and label it with coordinate values around the edge;
   - 1: in addition to the box and labels, draw the two axes (lines $x = 0$, $y = 0$) with tick marks;
   - 2: in addition to the box, labels, and axes, draw a grid at major increments of the x and y coordinates.

2.6 Labeling the Axes

Subroutine `PGLABEL` may (optionally) be called after `PGENV` to write identifying labels on the x and y axes, and at the top of the picture:

```
CALL PGLABEL('(x)', '(y)', 'A Simple Graph')
```

All three arguments are character variables or constants; any of them can be blank (`' '`).

1. A label for the x-axis (bottom of picture).
2. A label for the y-axis (left-hand edge).
3. A label for the plot (top of picture).

2.7 Drawing Graph Markers

Subroutine PGPOINT draws *graph markers* at one or more points on the graph. Here we use it to mark the five data points:

```
CALL PGPOINT(5,XS,YS,9)
```

If any of the specified points fall outside the window defined in the call to PGENV, they will not be plotted. The arguments to PGPOINT are:

1. The number of points to be marked (integer).

2, 3. The x and y coordinates of the points (real arrays).

4. The number of the symbol to be used to mark the points. In this example, we use symbol number 9 which is a circle with a central dot. The available symbols are shown in Chapter 4.

2.8 Drawing Lines

The following code draws the "theoretical curve" through the data points:

```
      DO 10 I=1,60
          XR(I) = 0.1*I
          YR(I) = XR(I)**2
   10 CONTINUE
      CALL PGLINE(60,XR,YR)
```

We compute the x and y coordinates at 60 points on the theoretical curve, and use subroutine PGLINE to draw a curve through them. PGLINE joins up the points with straight-line segments, so it is necessary to compute coordinates at fairly close intervals in order to get a smooth curve. Any lines which cross the boundary of the window defined in PGENV are "clipped" at the boundary, and lines which lie outside the boundary are not drawn. The arguments of PGLINE are like those of PGPOINT:

1. The number of points defining the line (integer).

2, 3. The x and y coordinates of the points (real arrays).

2.9 Ending the Plot

Subroutine PGEND must be called to complete the graph properly, otherwise some pending output may not get sent to the device:

```
CALL PGEND
```

2.10 Compiling and running the program

To compile the program and link it with the PGPLOT library, see Chapter 1. For example, under VMS:

```
$ EDIT SIMPLE.FOR
...
$ FORTRAN SIMPLE
$ LINK SIMPLE
```

Under Unix:

```
ed simple.f
...
fc -o simple simple.f -lpgplot
```

When you run the program, it will ask you to supply the graphics device specification. Type in any allowed device specification, or type a question-mark (`?`) to get a list of the available device types. For example, if you are using a VT125 terminal, type `/VT`: the graph will appear on the terminal screen.

If you want a hard copy, you can run the program again, and specify a different device type, e.g., `SIMPLE.PLT/VERS` to make a disk file in Versatec format. To obtain the hard copy, print the file (but first check with your system manager what the correct print command is; it is possible to waste a lot of paper by using the wrong command or sending a file to the wrong sort of printer!).

Chapter 3

WINDOWS AND VIEWPORTS

3.1 Introduction

This chapter is concerned with positioning a graph on the screen or hardcopy page, and controlling its scale. In simple applications, the position and scale of the graph are controlled more-or-less automatically by the routine `PGENV`, but in order to obtain complete control of positioning and scaling, it is necessary to understand the concepts of the *View Surface*, the *Window*, and the *Viewport*, and two coordinate systems: *World Coordinates* and *Device Coordinates*.

A simple PGPLOT picture might be a two-dimensional graph showing the dependence of one variable on another. A typical graph has data points, represented by error bars or special markers such as dots or diamonds, possibly connected by lines, or perhaps plotted on the same scale as a theoretical model drawn as a smooth curve. The graph must be labeled with two axes to indicate the coordinate scales.

The programmer must describe to PGPLOT the various elements of the graph in terms of rectangular Cartesian coordinates. The only limitation on the coordinates is that they be represented as floating-point (`REAL*4`) numbers; otherwise we are totally free to choose the meaning of the coordinates. For example, in a graph showing the temporal variation of a radio source, the abscissa (x-coordinate) might be Epoch (in years) and the ordinate (y-coordinate) Flux Density (in Jy).

In accordance with common practice in graphics programming, these coordinates, chosen by the programmer, are termed *world coordinates*. PGPLOT maps a selected rectangular region of the world-coordinate space (termed the *window*) onto a specified rectangle (termed the *viewport*) on the *view surface* (the screen of an interactive display or a sheet of paper on a hardcopy plotter). The program must make calls to PGPLOT routines to define both the window and the viewport. For complete descriptions of the routines and their arguments, refer to Appendix A.

3.2 Selecting a View Surface

The first thing a graphics program must do is to tell PGPLOT what device it is going to use. This is done by calling routine `PGBEGIN`. For example, to create a plot file for the Versatec printer:

```
CALL PGBEGIN (0, 'PLOTFILE.LIS/VERSATEC', 1, 1)
```

Equally important, when all plotting has been completed, it is necessary to call `PGEND` to flush any pending plot requests:

```
CALL PGEND
```

Note that only one device can be used at a time. If `PGBEGIN` is called while a plot is in progress, the old plot is closed and a new one is begun.

After calling `PGBEGIN` the program has access to a *view surface*. For interactive devices, this is the full screen of the device. For hardcopy devices, it is a standard page, usually $10(x) \times 8.5(y)$ inches on a device used in "landscape" mode (*e.g.*, device types `/VE` and `/QMS`), or $8.5(x) \times 10(y)$ inches a device used in "portrait" mode (*e.g.*, device types `/VV` and `/VQMS`).

On some devices, it is possible to plot on a larger piece of paper than the standard page; see the description of routine `PGPAPER`, which must be called immediately after `PGBEGIN` to change the size of the view surface. The different devices differ not only in the size of the view surface, but also in its *aspect ratio* (height/width). `PGPAPER` can be called to ensure that a plot has the same aspect ratio no matter what device it is plotted on.

After completing a graph, it is possible to advance to a new page to start a new graph (without closing the plot file) by calling `PGPAGE`:

```
CALL PGPAGE
```

This clears the screen on interactive devices, or gives a new piece of paper on hardcopy devices. It does not change the viewport or window.

The last two arguments of `PGBEGIN` (`NX` and `NY`) can be used to subdivide the view surface into smaller pieces called sub-pages, each of which can then be used separately. The view-surface is divided into NX (horizontally) by NY (vertically) sub-pages. When the view surface has been subdivided in this way, `PGPAGE` moves the plotter to the next sub-page, and only clears the screen or loads a new piece of paper if there are no sub-pages left on the current page.

In addition to selecting the view surface, `PGBEGIN` also defines a default viewport and window. It is good practice, however, to define the viewport and window explicitly as described below.

3.3 Defining the Viewport

A *viewport* is a rectangular portion of the plotting surface onto which the graph is mapped. PGPLOT has a default viewport which is centered on the plotting surface and leaves sufficient space around it for annotation. The application program can redefine the viewport by calling routine `PGVPORT` or `PGVSIZE`.

`PGVPORT` defines the viewport in a device-independent manner, using a coordinate system whose coordinates run from 0 to 1 in both x and y. This coordinate system is called *normalized device coordinate space*. For example, if we wish to divide the view surface into four quadrants and map a different plot onto each quadrant, we can define a new viewport before starting each plot. `PGVPORT` has the format:

```
CALL PGVPORT (XMIN, XMAX, YMIN, YMAX)
```

For example, to map the viewport onto the upper left quadrant of the view surface:

```
CALL PGVPORT (0.0, 0.5, 0.5, 1.0)
```

(Note that this does not leave room around the edge of the viewport for annotation.)

`PGVSIZE` defines the viewport in absolute coordinates (inches); it should only be used when it is known how big the view surface is and a definite plot scale is required. The arguments are the same as for `PGVPORT`, but measured in inches from the bottom left corner of the view surface. For example:

```
CALL PGVSIZE (1.5, 9.5, 1.5, 6.5)
```

defines a rectangular viewport 8 by 5 inches, offset 1.5 inches from the bottom and left edges of the view surface.

`PGVSTAND` defines a standard viewport, the size of which depends on the particular device being used, and on the current character size (it uses the whole view surface excluding a margin of four character heights all around):

```
CALL PGVSTAND
```

This is the default viewport set up by `PGBEGIN`.

Note that the viewport must be defined *before* calling any routines that would actually generate a display. The viewport may, however, be changed at any time: this will affect the appearance of objects drawn later in the program.

3.4 Defining the Window

The program defines the *window* by calling routine PGWINDOW, whose arguments specify the world-coordinate limits of the window along each coordinate axis. *e.g.*:

```
CALL PGWINDOW (1975.0, 1984.0, 5.0, 20.0)
```

specifies that the x-axis (epoch) is going to run (left to right) from 1975 to 1984, and the y-axis (flux density) is going to run (bottom to top) from 5 to 20 Jy. Note that the arguments are floating-point numbers (Fortran REAL variables or constants), and require decimal points. If the order of either the x pair or the y pair is reversed, the corresponding axis will point in the opposite sense, *i.e.*, right to left for x or top to bottom for y. PGPLOT uses the window specification to construct a mapping that causes the image of the window to coincide with the viewport on the view surface. Furthermore, PGPLOT "clips" lines so that only those portions of objects that lie within the window are displayed on the view surface.

Like the viewport, the window must be defined before drawing any objects. The window can be defined either before or after the viewport: the effect will be the same. The default window, set up by PGBEGIN, has x limits 0.0–1.0 and y limits 0.0–1.0.

If the ratio of the sides of the window does not equal the ratio of the sides of the viewport, the mapping of the world coordinates onto the view surface results in an image whose shape is compressed in either x or y. One way to avoid this compression is to carefully choose the viewport to have the same aspect ratio as the window. Routine PGWNAD can do this: it defines the window and simultaneously adjusts the viewport to have the same aspect ratio as the window. The new viewport is the largest that can fit inside the old one, and is centered in the old one.

3.5 Annotating the Viewport

For a simple graph, it is usually necessary to draw a frame around the viewport and label the frame with tick marks and numeric labels. This can be done with the routine PGBOX. For our sample graph, the call might be:

```
CALL PGBOX ('BCTN', 0.0, 0, 'BCNST', 0.0, 0)
```

Another routine, PGLABEL, provides text labels for the bottom, left hand side, and top of the viewport:

```
CALL PGLABEL ('Epoch', 'Flux Density (Jy)',
              'Variation of 3C345 at 10.7 GHz')
```

The first two arguments provide explanations for the two axes; the third provides a title for the whole plot. Note that unlike all the other plotting routines, the lines and characters drawn by PGBOX and PGLABEL are not clipped at the boundaries of the window. PGLABEL actually calls a more general routine, PGMTEXT, which can be used for plotting labels at any point relative to the viewport.

The amount of space needed outside the viewport for annotation depends on the exact options specified in PGBOX; usually four character heights will be sufficient, and this is the amount allowed when the standard viewport (created by PGVSTAND) is used. The character height can be changed by using routine PGSCH.

3.6 Routine PGENV

Having to specify calls to PGPAGE, PGVPORT, PGWINDOW, and PGBOX is excessively cumbersome for drawing simple graphs. Routine PGENV (for PGplot ENVironment) combines all four of these in one subroutine, using the standard viewport, and a limited set of the capabilities of PGBOX. For example, the graph described above could be initiated by the following call:

```
CALL PGENV (1975.0, 1984.0, 5.0, 20.0, 0, 0)
```

which is equivalent to the following series of calls:

```
CALL PGPAGE
CALL PGVSTAND
CALL PGWINDOW (1975.0, 1984.0, 5.0, 20.0)
CALL PGBOX ('BCNST', 0.0, 0, 'BCNST', 0.0, 0)
```

PGENV uses the standard viewport. The first four arguments define the world-coordinate limits of the window. The fifth argument can be 0 or 1; it is 1, PGENV calls PGWNAD instead of PGWINDOW so that the plot has equal scales in x and y. The sixth argument controls the amount of annotation.

Chapter 4

PRIMITIVES

4.1 Introduction

Having selected a view surface and defined the viewport and the window, we are ready to draw the substance of the image that is to appear within the viewport. This chapter describes the most basic routines, called *primitives*, that can be used for drawing elements of the image. There are four different sorts of primitive: *lines, graph-markers, text*, and *area fill.* Chapter 5 explains how to change the *attributes* of these primitives, *e.g.*, color, line-style, text font; and Chapter 6 describes some higher-level routines that simplify the composition of images that would require a large number of calls to the primitive routines.

The primitive routines can be used in any combination and order after the viewport and window have been defined. They all indicate where the primitive is to appear on the view surface by specifying world coordinates. See the subroutine descriptions in Appendix A for more details.

4.2 Clipping

The primitives are "clipped" at the edge of the viewport: any parts of the image that would appear outside the viewport are suppressed. The various primitives behave slightly differently. A *line* is clipped where it crosses the edge of the viewport. A *graph marker* is plotted if the center (the point marked) lies within or on the edge of the viewport; otherwise it is suppressed. *Text*, which is usually used for annotation, is not clipped (except at the edge of the view surface. A *filled area* is clipped at the edge of the viewport.

4.3 Lines

The primitive line-drawing routine is PGLINE. This draws one or more connected straight-line segments (generally called a *polyline* in computer graphics). It has three arguments: the number (N) of points defining the polyline, and two arrays (XPTS and YPTS) containing the world x and y-coordinates of the points. The polyline consists of $N-1$ straight-line segments connecting points 1–2, 2–3, ..., $(N-1)$–N:

```
CALL PGLINE (N, XPTS, YPTS)
```

The two routines PGMOVE and PGDRAW are even more primitive than PGLINE, in the sense that any line graph can be produced by calling these two routines alone. In general, PGLINE should be preferred, as it is more modular. PGMOVE and PGDRAW are provided for those who are used to Calcomp-style plotting packages. PGMOVE moves the plotter "pen" to a specified point, without drawing a line ("pen up"). It has two arguments: the world-coordinates of the required new pen position. PGDRAW moves the plotter "pen" from its current position (defined by the last call of PGMOVE or PGDRAW) to a new point, drawing a straight line as it goes ("pen down"). The above call to PGLINE could be replaced by the following:

```
CALL PGMOVE (XPTS(1), YPTS(1))
DO I=2,N
    CALL PGDRAW (XPTS(I), YPTS(I))
END DO
```

4.4 Graph Markers

A Graph Marker is a symbol, such as a cross, dot, or circle, drawn on a graph to mark a specific point. Usually the symbol used will be chosen to be symmetrical with a well-defined center. The routine PGPOINT draws one or more graph markers (sometimes called a *polymarker*). It has four arguments: the number (N) of points to mark, two arrays (XPTS and YPTS) containing the world x and y-coordinates of the points, and a number (NSYM) identifying the symbol to use:

```
CALL PGPOINT (N, XPTS, YPTS, NSYM)
```

The symbol number can be: -1, to draw a dot of the smallest possible size (one pixel); 0–31, to draw any one of the symbols in Figure 4.1; or 33–127, to draw the corresponding ASCII character (the character is taken from the currently selected text font); or > 127, to draw one of the Hershey symbols from Appendix B. The Fortran ICHAR function can be used to obtain the ASCII value; *e.g.*, to use letter F:

```
CALL PGPOINT (1, 0.5, 0.75, ICHAR('F') )
```

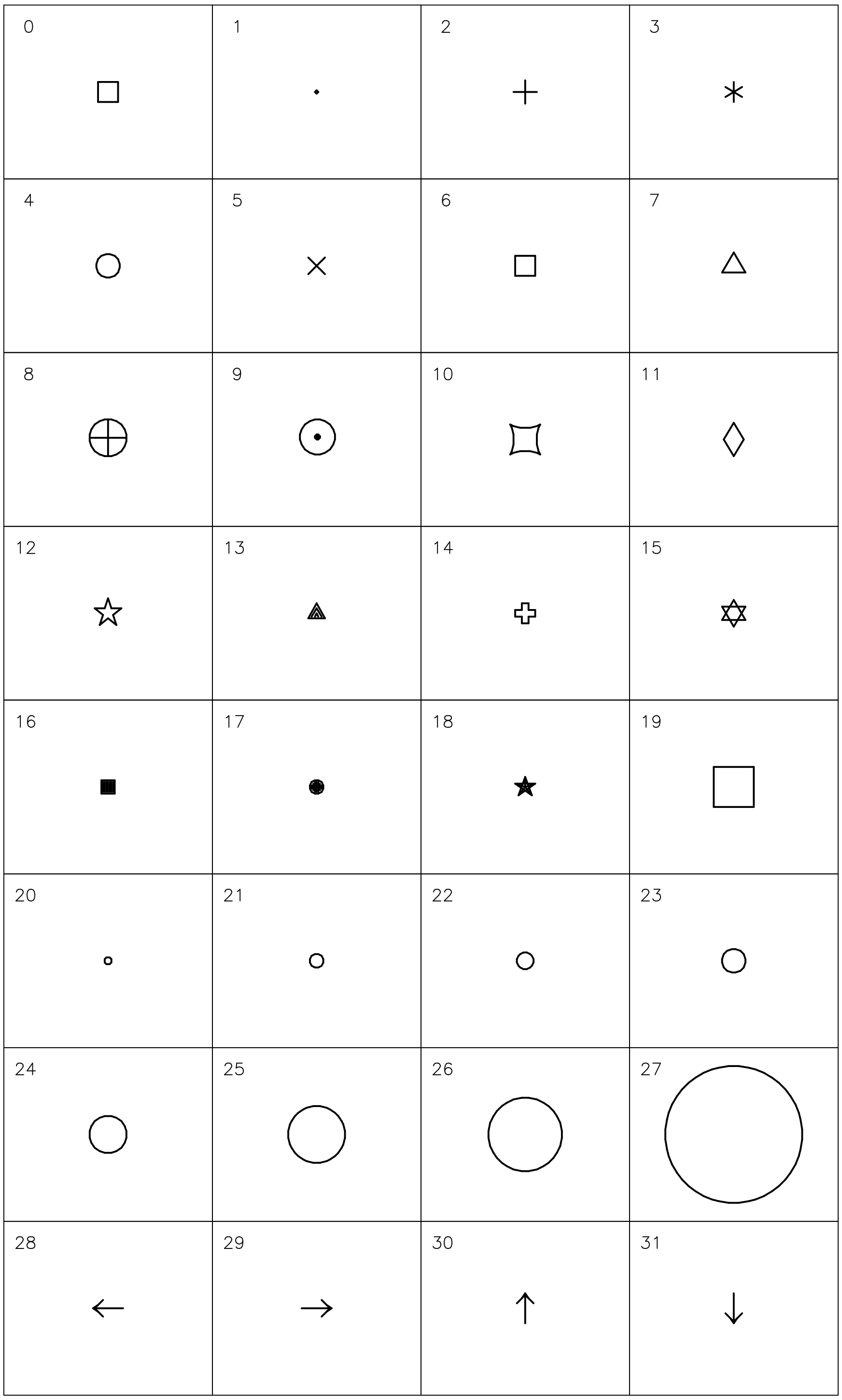

Figure 4.1 Standard Graph Markers.

4.5 Text

The Text primitive routine is used for writing labels and titles on the image. It converts an internal computer representation of the text (ASCII codes) into readable text. The simplest routine for writing text is `PGTEXT`, which writes a horizontal character string starting at a specific (x, y) world coordinate position, *e.g.*,

```
CALL PGTEXT (X, Y, 'A text string')
```

`PGTEXT` is actually a simplified interface to the more general primitive routine `PGPTEXT`, which allows one to change orientation and justification of the text, *e.g.*,

```
CALL PGPTEXT (X, Y, 45.0, 0.5, 'A text string')
```

writes the text at an angle of 45° to the horizontal, centered at (x, y) (see Appendix A).

Both `PGTEXT` and `PGMTEXT` require the position of the text string to be specified in world coordinates. When annotating a graph, it is usually more convenient to position the text relative to the edge of the viewport, rather than in world-coordinate space. The routine `PGMTEXT` (see Appendix A) is provided for this, and `PGLABEL` provides a simple interface to `PGMTEXT` for the normal job of annotating an (x, y) graph.

The appearance of text can be altered by specifying a number of *attributes*, described in the next chapter. In particular, the character size and character font can be changed. Figure 4.2 illustrates some of the possibilities.

To include one of the graph marker symbols in a text string, use the Fortran `CHAR` function, *e.g.*,

```
CALL PGTEXT (X, Y, 'Points marked with '//CHAR(17))
```

The routine `PGPTEXT` (and all the PGPLOT routines which call it, *e.g.*, `PGTEXT`, `PGLABEL`) allows one to include *escape sequences* in the text string to be plotted. These are character-sequences that are not plotted, but are interpreted as instructions to change font, draw superscripts or subscripts, draw non-ASCII characters (*e.g.*, Greek letters), etc. All escape sequences start with a backslash character (`\`). The following escape sequences are defined (the letter following the `\` may be either upper or lower case):

`\u` – start a superscript, or end a subscript;

`\d` – start a subscript, or end a superscript (note that `\u` and `\d` must always be used in pairs);

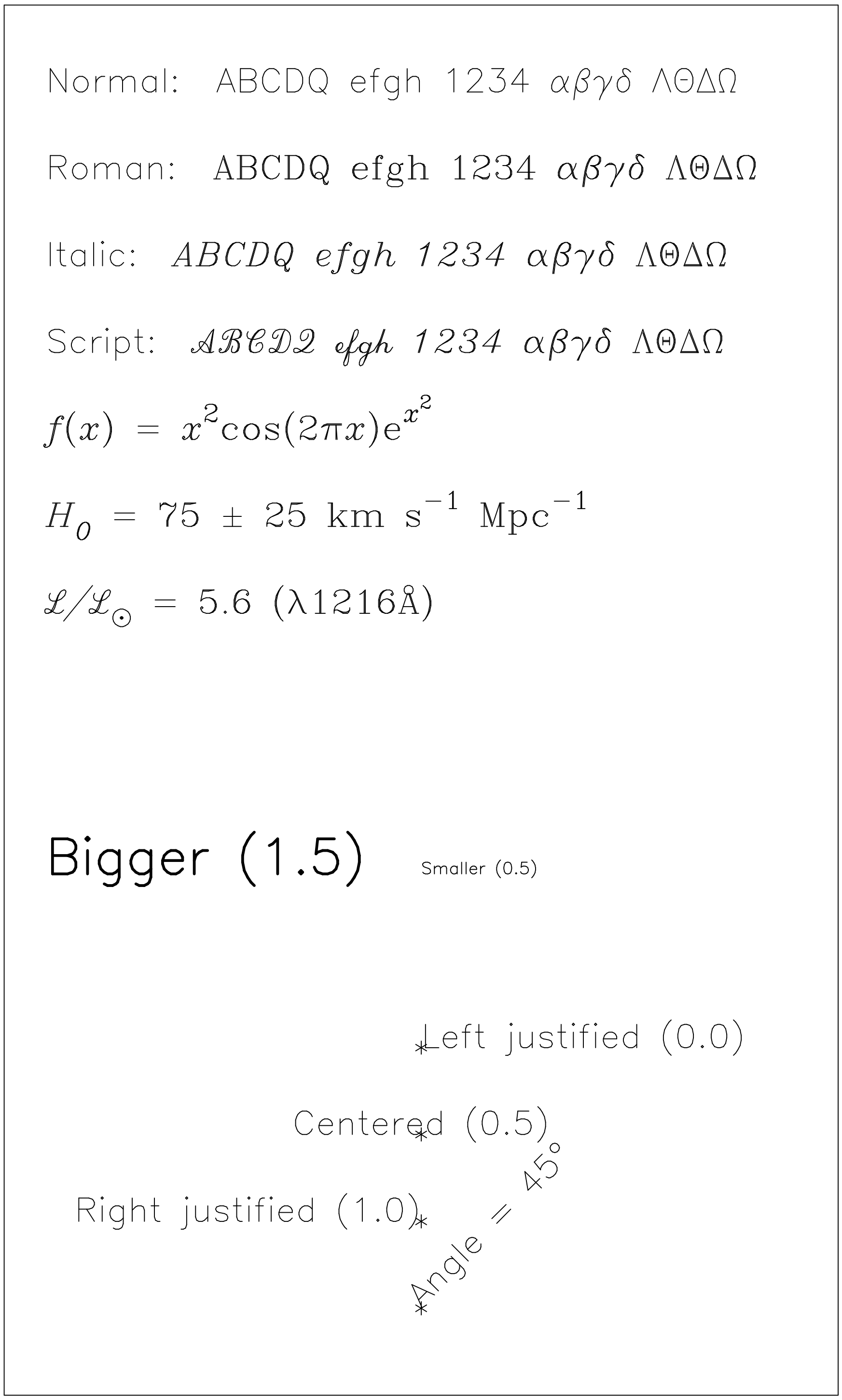

Figure 4.2 Text examples.

`\b` – backspace (*i.e.*, do not advance text pointer after plotting the previous character);

`\\` – backslash character (`\`);

`\A` – Ångström symbol (Å);

`\g`x – greek letter corresponding to roman letter x;

`\fn` – switch to Normal font (1);

`\fr` – switch to Roman font (2);

`\fi` – switch to Italic font (3);

`\fs` – switch to Script font (4);

`\(`n`)` – character number n (1 to 4 decimal digits); the closing parenthesis may be omitted if the next character is neither a digit nor ")". This makes a number of special characters (*e.g.*, mathematical, musical, astronomical, and cartographical symbols) available. See Appendix B for a list of available characters.

Greek letters are obtained by `\g` followed by one of the following upper-case and lower-case letters:

use:	A	B	G	D	E	Z	Y	H	I	K	L	M	N	C	O	P	R	S	T	U	F	X	Q	W
for:	A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M	N	Ξ	O	Π	P	Σ	T	Υ	Φ	X	Ψ	Ω
or:	a	b	g	d	e	z	y	h	i	k	l	m	n	c	o	p	r	s	t	u	f	x	q	w
for:	α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ	ν	ξ	o	π	ρ	σ	τ	υ	ϕ	χ	ψ	ω

Use uppercase letters for uppercase Greek, lowercase for lowercase. Example: `\gh` is θ, lowercase "theta".

4.6 Area Fill

The Area Fill primitive allows the programmer to shade the interior of an arbitrary polygonal region. The appearance of the primitive is controlled by attributes *fill area style* and *color index* (see Chapter 5). An area is specified by the set of vertices of the polygon.

The routine `PGPOLY` is used to fill an area. It has three arguments: the number (`N`) of vertices defining the polygon, and two arrays (`XPTS` and `YPTS`) containing the world x and y-coordinates of the vertices:

```
CALL PGPOLY (N, XPTS, YPTS)
```

If the polygon is not convex, it may not be obvious which points in the image are inside the polygon. PGPLOT assumes that a point is inside the polygon if a straight line drawn from the point to infinity intersects an odd number of the polygon's edges.

For the special case of a *rectangle* with edges parallel to the coordinates axes, it is better to use routine PGRECT instead of PGPOLY; this routine will use the hardware rectangle-fill capability if available. PGRECT has four arguments: the (x, y) world coordinates of two opposite corners (note the order of the arguments):

```
CALL PGRECT (X1, X2, Y1, Y2)
```

Chapter 5

ATTRIBUTES

5.1 Introduction

The appearance of the primitive elements of a graphical image (lines, graph-markers, text, and area-fill) can be changed by specifying *primitive attributes.* The attributes, and the corresponding routines for changing them, are:

Color Index and *Color Representation:* `PGSCI`, `PGSCR`, and `PGSHLS`.

Line Style: `PGSLS`.

Line Width: `PGSLW`.

Character Height: `PGSCH`.

Character Font: `PGSCF`.

Fill-area Style: `PGSFS`.

The routines to change attributes can be freely intermixed with the PGPLOT drawing routines. Once an attribute has been changed by a call to the appropriate routine, it remains in effect for all subsequent plotting until it is changed again. In addition to the routines that set attributes (`PGSxx`) there are routines for determining the current value of each attribute (`PGQxx`). These make it possible to write subroutines which change attribute values temporarily but restore the old attributes before returning to the calling program.

5.2 Color Index

This attribute affects all the primitives: lines, graph-markers, text, and area-fill, and is controlled by two subroutines: `PGSCI` and `PGSCR`.

Devices differ considerably in their ability to draw in more than one color. On most hardcopy devices, the default color is black on a white background, while on most CRT devices, it is white (or green) on a black background. Color is selected using an integer parameter called the *color index.* Color index 1 is the default color, and color index 0 is the background color. The number of different color indices available depends on the device. On most monochrome devices, only color indices 0 and 1 are available, while some color CRT devices may permit color indices from 0 to 255. On some monochrome devices, color index can be used to select different brightnesses (intensities).

Color index 0, the background color, can be used to "erase" elements from a picture by overwriting them with color index 0. Note that not all devices are capable of this: e.g., Tektronix storage-tube terminals and pen-plotters cannot erase part of a displayed picture.

To select a new color index for subsequent plotting, use routine `PGSCI` (Set Color Index).

Appendix D lists the capabilities of the devices for plotting in color and variable intensity. The default color index is 1; all devices accept this. Most devices also accept color index 0 (erase), and several accept color index up to 15 or more. The maximum color index is the number of different colors that can be displayed at once. Some devices permit the assignment of colors to color indices to be changed (by calling `PGSCR`, see below).

5.3 Color Representation

Each color index has an associated *Color Representation*, which defines the associated color and intensity. Color Representation may be expressed by a set of three numbers, either the Hue, Lightness, and Saturation (H, L, S) components or the Red, Green, and Blue (R, G, B) components. (R, G, B) are quantities in the range 0.0–1.0, with 1.0 being maximum intensity; if $R = G = B$ the color is a shade of gray. In the (H, L, S) system, hue is a cyclic quantity expressed as an angle in the range 0–360, while L and S are in the range 0.0–1.0.

Table 5.1 shows how the color indices are defined when PGPLOT is started (not all are available on all devices). The default assignments of colors to color indices can be changed with routine `PGSCR`, which permits one to specify the (R, G, B) values for any color index, or `PGSHLS`, which permits one to specify the (H, L, S) values. Note that color-index 0, the background color, can be redefined in this way.

The effect of `PGSCR` is device-dependent. One some devices, it will be ignored. On others, (*e.g.*, Grinnell, VT125) it will change the color of lines which have already been drawn in the specified color index, while on others (*e.g.*, pen plotters) it will only affect lines drawn after the call of `PGSCR`.

Table 5.1 Default Color Representations

Color Index	*Color*	(H, L, S)	(R, G, B)
0	Black (background)	0, 0.00, 0.00	0.00, 0.00, 0.00
1	White (default)	0, 1.00, 0.00	1.00, 1.00, 1.00
2	Red	120, 0.50, 1.00	1.00, 0.00, 0.00
3	Green	240, 0.50, 1.00	0.00, 1.00, 0.00
4	Blue	0, 0.50, 1.00	0.00, 0.00, 1.00
5	Cyan (Green + Blue)	300, 0.50, 1.00	0.00, 1.00, 1.00
6	Magenta (Red + Blue)	60, 0.50, 1.00	1.00, 0.00, 1.00
7	Yellow (Red + Green)	180, 0.50, 1.00	1.00, 1.00, 0.00
8	Red + Yellow (Orange)	150, 0.50, 1.00	1.00, 0.50, 0.00
9	Green + Yellow	210, 0.50, 1.00	0.50, 1.00, 0.00
10	Green + Cyan	270, 0.50, 1.00	0.00, 1.00, 0.50
11	Blue + Cyan	330, 0.50, 1.00	0.00, 0.50, 1.00
12	Blue + Magenta	30, 0.50, 1.00	0.50, 0.00, 1.00
13	Red + Magenta	90, 0.50, 1.00	1.00, 0.00, 0.50
14	Dark Gray	0, 0.33, 0.00	0.33, 0.33, 0.33
15	Light Gray	0, 0.66, 0.00	0.66, 0.66, 0.66
16–255	Undefined		

5.4 Line Style

Line Style can be, *e.g.*, solid, dashed, or dotted. The attribute affects only lines, not the other primitives. It is controlled by subroutine `PGSLS`. The default line style is a full, unbroken line. To change the line style, use routine `PGSLS`. Line style is described by an integer code:

1 – full line,

2 – long dashes,

3 – dash-dot-dash-dot,

4 – dotted,

5 – dash-dot-dot-dot.

5.5 Line Width

Line Width affects lines, graph-markers, and text. A thick-nibbed pen is simulated by tracing each line with multiple strokes offset in the direction perpendicular to the line. The line width is specified by the number of strokes. The default width is one stroke, and the maximum that may be specified is 201. The exact appearance of thick lines is device-dependent—it depends on the resolution of the device—but on hardcopy devices (*e.g.*, QMS Lasergrafix, Versatec) PGPLOT attempts to make the line-width increment equal to 0.005 inches. Requesting a line-width of 10, say, should give lines that are approximately 1/20 inch thick. To change the line width, use routine `PGSLW`.

5.6 Character Height

Character Height affects graph-markers and text. Character height is specified as a multiple of the default character height; the default character height one-fortieth of the height or width of the view surface (whichever is less). To change the character height, use routine `PGSCH`.

5.7 Character Font

Character Font affects text only. Four fonts are available. The default font (1) is simple and is the fastest to draw; the others should only be used for presentation plots on a high-resolution device (*e.g.*, Versatec or laser printer). To change the character font, use routine `PGSCF`; it is also possible to change the font temporarily by using escape sequences (see §4.4). The font is defined by an integer code:

1 – normal (simple) font (default),

2 – roman font,

3 – italic font,

4 – script font.

5.8 Fill-Area Style

Fill-Area Style can be hollow (only the outline of the polygon is drawn), or solid. The attribute may be extended in future to allow hatching and other patterns. To change the fill-are style, use routine `PGSFS`. The style is defined by an integer code:

1 – solid fill (default),

2 – hollow (outline only).

Chapter 6

HIGHER-LEVEL ROUTINES

6.1 Introduction

This chapter describes a number of "high level" routines that simplify the composition of complicated graphical images. They include routines for drawing graphs of one variable or function against another ("xy-plots"), histograms, and display of two-dimensional data (functions of two variables). Rather than giving complete details of all the available routines, this chapter just points out some of the ways that they can be used. See Appendix A for details of the calling sequences.

6.2 XY-plots

The basic technique for drawing xy-plots is described in Chapter 2, which showed how to make *scatter plots* using graph markers produced by PGPOINT and *line plots* produced by PGLINE. Considerable variation in the appearance of the graph can be achieved using the following techniques.

Attributes. Use different attributes to distinguish different datasets. Graph markers can be distinguished by choosing different markers, different colors, or different sizes (character height attribute). Lines and curves can be distinguished by line-style, color, or line-width.

Box parameters. If routine PGENV is replaced by calls to the more basic routines (see §3.6), including PGBOX, considerable variety in the appearance of the graph can be achieved. For example, one can suppress the tick marks, draw the tick marks projecting out of the box instead of into it, or draw a grid over the whole viewport. Note that PGBOX may be called many times: one might call it once to draw a grid using thin dotted lines, and again to draw the frame and tick marks using thick lines:

```
CALL PGSLW(1)
CALL PGSLS(4)
CALL PGBOX('G',30.0,0,'G',0.2,0)
CALL PGSLW(3)
CALL PGSLS(1)
CALL PGBOX('ABCTSN',90.0,3,'ABCTSNV',0.0,0)
```

Note that in this example we have also specified tick intervals explicitly. If the horizontal axis is to represent an angle in degrees, it is convenient to choose a tick interval that is a simple fraction of 360; here we have a major tick interval of 90° and a minor tick interval of 30°.

Stepped-line plots. As an alternative to `PGLINE`, which "joins up the dots" using straight line segments, it is sometimes appropriate to use `PGBIN` which produces a "stepped line plot" (sometimes misleadingly called a histogram) with horizontal line segments at each data point and vertical line segments joining them. This is often used, for example, in displaying digitized spectra.

Error bars. Graphs of real data often require the inclusion of error bars. The two routines `PGERRX` and `PGERRY` draw horizontal and vertical error bars, respectively. These routines are usually used in combination with `PGPOINT`, *e.g.*, to draw a set of points with $\pm 2\sigma$ error-bars:

```
      DO 10 I=1,15
          YHI = YPTS(I) + 2.0*ERR(I)
          YLO = YPTS(I) - 2.0*ERR(I)
          CALL PGPOINT(1, XPTS(I), YPTS(I), 17)
          CALL PGERRY(1, XPTS(I), YLO, YHI, 1.0)
   10 CONTINUE
```

Logarithmic axes. It is commonly required that the x-axis, the y-axis, or both, be logarithmic instead of linear; that is, one wishes to plot the logarithm of the quantity instead of its actual value. PGPLOT doesn't provide any automatic mechanism to do this: one has to adopt $\log_{10} x$ and/or $\log_{10} y$ instead of x and y as world-coordinates; *i.e.*, if the range of x is to be 1 to 1000, choose as world-coordinate limits for the window $\log 1 = 0.0$ and $\log 1000 = 3.0$, and supply the logarithms of x to `PGPOINT` and `PGLINE`. However, `PGENV` and `PGBOX` have options for *labeling* the axis logarithmically; if this option is used in our example, the axis will have labeled major tick marks at 1, 10, 100, and 1000, with logarithmically-spaced minor tick marks at 2, 3, 4, ..., 20, 30, 40, etc.. An example may make this clearer:

```
      CALL PGENV(-2.0,2.0,-0.5,2.5,1,30)
      CALL PGLABEL('Frequency, \gn (GHz)',
     1             'Flux Density, S\d\gn\u (Jy)', ' ')
      DO 10 I=1,15
          XPTS(I) = ALOG10(FREQ(I))
          YPTS(I) = ALOG10(FLUX(I))
   10 CONTINUE
      CALL PGPOINT(15, XPTS, YPTS, 17)
```

This is a fragment of a program to draw the spectrum of a radio source, which is usually plotted as a log–log plot of flux density v. frequency. It first calls `PGENV` to initialize the viewport and window; the `AXIS` argument is 30 so both axes will be logarithmic. The x-axis (frequency, ν) runs from 0.01 to 100 GHz, the y-axis (flux density, S_ν) runs from 0.3 to 300 Jy. Note that it is necessary to specify the logarithms of these limits in the call to `PGENV`. The penultimate argument requests equal scales in x and y so that slopes will be correct. The program then marks 15 data points, supplying the *logarithms* of frequency and flux density to `PGPOINT`.

6.3 Histograms

The routine `PGHIST` draws a histogram, that is, the frequency distribution of measured values in a dataset. Suppose we have 500 measurements of a quantity (the sky brightness temperature at 20 GHz, say, in mK) stored in Fortran array `VALUES`. The following program-fragment draws a histogram of the distribution of these values in the range 0.0 to 5.0, using 25 bins (so that each bin is 0.2 K wide, the first running from 0.0 to 0.2, the second from 0.2 to 0.4, etc.):

```
      DO 10 I=1,500
          VALUES(I) = ....
   10 CONTINUE
      CALL PGHIST(500, VALUES, 0.00, 5.00, 25, 0)
      CALL PGLABEL('Temperature (K)',
     1             'Number of measurements',
     2             'Sky Brightness at 20 GHz' )
```

The histogram does not depend on the *order* of the values within the array.

6.4 Functions of two variables

A function of two variables, $f = f(x, y)$, really needs a three-dimensional display. `PGPLOT` does not have any three-dimensional display capability, but it provides three methods for two-dimensional display of three-dimensional data.

Contour maps. In a contour map of $f(x,y)$, the world-coordinates are x and y and the contours are lines of constant f. The PGPLOT contouring routines (`PGCONT` and `PGCONS`) require the input data to be stored in a two-dimensional Fortran array `F`, with element `F(I,J)` containing the value of the function $f(x,y)$ for a point (x_i, y_j). Furthermore, the function must be sampled on a regular grid: the (x,y) coordinates corresponding to `(I,J)` must be related to `I` and `J` by:

$$x = a + b\mathtt{I} + c\mathtt{J},$$
$$y = d + e\mathtt{I} + f\mathtt{J}.$$

The constants a, b, c, d, e, f are supplied to `PGCONT` in a six-element Fortran array. The other input required is an array containing the contour values, *i.e.*, the constant values of f corresponding to each contour to be drawn. In the following example, we assume that values have been assigned to the elements of array `F`. We first find the maximum and minimum values of `F`, and choose 10 contour levels evenly spaced between the maximum and minimum:

```
      REAL F(50,50), ALEV(10), TR(6)
      ...
      FMIN = F(1,1)
      FMAX = F(1,1)
      DO 300 I=1,50
        DO 200 J=1,50
          FMIN = MIN(F(I,J),FMIN)
          FMAX = MAX(F(I,J),FMAX)
  200   CONTINUE
  300 CONTINUE
      DO 400 I=1,10
        ALEV(I) = FMIN + (I-1)*(FMAX-FMIN)/9.0
  400 CONTINUE
```

Next, we choose a window and viewport, and set up an array `TR` containing the 6 constants in the transformation between array indices and world coordinates. In this case, the transformation is simple, as we want $x =$ `I`, $y =$ `J`:

```
      CALL PGENV(0.,50.,5.,45.,0,2)
      TR(1) = 0.0
      TR(2) = 1.0
      TR(3) = 0.0
      TR(4) = 0.0
      TR(5) = 0.0
      TR(6) = 1.0
```

Finally, we call PGCONT; actually, we call it twice, to draw the first five contours in color index 2 (red) and the remaining 5 in color index 3 (green):

```
CALL PGSCI(2)
CALL PGCONT(F,50,50,1,50,1,50,ALEV,5,TR)
CALL PGSCI(3)
CALL PGCONT(F,50,50,1,50,1,50,ALEV(6),5,TR)
```

Normally PGCONT is preferable to PGCONS. See the description in Appendix A for suggestions as to when PGCONS should be used.

Gray-scale plots. The routine PGGRAY is used in a similar way to PGCONT. Instead of drawing contours, it shades the interior of the viewport, the intensity of shading representing the value of the function at each point. The exact appearance of the resulting "image" is device-dependent. On some devices, PGGRAY does the shading by drawing many dots, so it can be *very* slow.

Cross sections. Routine PGHI2D draws a series of cross-sections through a two-dimensional data array. Each cross-section "hides" those that appear behind it, giving a three-dimensional effect. See Appendix A for details.

Chapter 7

INTERACTIVE GRAPHICS

7.1 Introduction

The previous chapters have described how to produce a *static* graphical image: if the same program is run twice with the same input parameters, the same image will result. An *interactive* program allows the user to control the behavior of the program with a graphical input device. PGPLOT supports a limited interactive capability on devices with a cursor for graphical input (*e.g.*, Grinnell, VT125 terminal, some Tektronix terminals, VT640 Retrographics terminal). The capabilities are necessarily limited by the aim to keep PGPLOT device-independent.

7.2 The Cursor

Some of the graphics devices supported by PGPLOT have a *graphics cursor*. This appears on the view surface as a plus sign, a cross-hair, or a diamond, and can be moved around the view surface with a joy-stick, mouse, or trackball attached to the graphics device. If the hardware does not provide this mechanism, PGPLOT allows the user to move the cursor using the arrow keys on his terminal. See Appendix D for instructions for using the cursor on a specific device.

7.3 Using the Cursor

The basic routine for cursor input is `PGCURSE`. This routine enables the cursor on the selected device, positions it at a specified location within the viewport, and allows the user to move it. When the user has positioned the cursor, he types a key on his terminal; `PGCURSE` returns the cursor position (in world coordinates) and the character that was typed.

In addition, PGPLOT provides three higher-level routines for cursor input: `PGOLIN`, `PGNCURSE`, and `PGLCUR`. These three routines require that the device has erase capability.

PGOLIN allows the user to specify a set of points within the viewport, with the capability of correcting mistakes. Interactive commands (single characters [A, D, or X] typed on the keyboard) allow the user to *add* a point at the current cursor position, *delete* the last-entered point, or *exit* from the subroutine. The world-coordinates of the entered points are returned to the calling program. The following program fragment illustrates the use of PGOLIN; the user supplies NPT (up to 50) points with world-coordinates X() and Y(), and the program then shades the polygon defined by these points by calling PGPOLY:

```
INTEGER NPT
REAL X(50), Y(50)
...
WRITE (6,*) 'Use the cursor to draw a polygon'
WRITE (6,*) 'Type A to add point, D to delete, X to exit'
NPT = 0
CALL PGOLIN (50, NPT, X, Y, 0)
IF (NPT.GE.3) CALL PGPOLY (NPT, X, Y)
```

PGNCURSE is similar to PGOLIN, but the points are sorted into increasing order of x before being returned to the calling program. In addition, the *delete* command deletes the point nearest to the cursor, rather than the last-entered point. It can be used, for example, to allow the user to supply a set of points to represent the continuum level on a spectrum.

PGLCUR is similar to PGOLIN but instead of using a graph marker to mark each entered point it draws a polyline through them.

7.4 Buffering

By default, PGPLOT ensures that the image seen on the view surface is up to date at all times; that is, each PGPLOT subroutine updates the image before returning control to the calling program. To improve efficiency, PGPLOT can save instructions for controlling the graphics device in a buffer, and only send them to the device when the buffer is filled up. This means that at any given moment, the image displayed on the screen may not be completely up to date. This can be a problem in an interactive program, where, for example, the user has to tell the program what to do next based on his interpretation of the current display. Three PGPLOT routines (PGBBUF, PGEBUF, and PGUPDT) are provided for controlling the buffering of output. All three routines have no arguments.

The routine PGBBUF causes PGPLOT to begin saving graphical output in a buffer. The output is saved until (1) a matching PGEBUF call is made, or (2) the buffer fills up, or (3) the buffer is emptied by a call to PGUPDT, or (4) PGEND is called. The routine PGEBUF stops buffering and causes the buffered commands to be sent to the output device. Calls to PGBBUF and PGEBUF should always be paired. PGBBUF increments an internal counter, while PGEBUF decrements this counter and flushes the buffer to the output device when the counter drops to zero. This allows a subroutine to turn on and turn off buffering without disturbing any buffering that may have been established by the calling program.

Routine PGUPDT empties the buffer created by PGBBUF, but it does not alter the internal counter. The routine should be called when it is essential that the display be completely up-to-date (before interaction with the user, for example) but it is not known if output is being buffered.

Usually output is not buffered; this is the default state established by PGBEGIN. The default behavior can be changed, however, by defining an environment variable PGPLOT_BUFFER (see Chapter 1). If this variable is defined, with any value, PGBEGIN will start buffering output (by calling PGBBUF).

The following example shows how routine PGLABEL might be implemented in terms of routine PGMTEXT:

```
SUBROUTINE PGLABEL (XLBL, YLBL, TOPLBL)
CHARACTER*(*) XLBL, YLBL, TOPLBL
CALL PGBBUF
CALL PGMTEXT('T', 2.0, 0.5, 0.5, TOPLBL)
CALL PGMTEXT('B', 3.2, 0.5, 0.5, XLBL)
CALL PGMTEXT('L', 2.2, 0.5, 0.5, YLBL)
CALL PGEBUF
END
```

The calls to PGBBUF and PGEBUF ensure that the output generated by the three calls to PGMTEXT is buffered (i.e., sent to the output device as a single command instead of three separate ones). If buffering is already enabled by the program which calls PGLABEL, the calls to PGBBUF and PGEBUF have no effect.

Chapter 8

METAFILES

8.1 Introduction

A graphics *metafile* is a disk file in which a device-independent representation of a graphics image can be stored. Such a file cannot be displayed directly on a graphics device, but must first be translated into the commands appropriate for the specific device using a *metafile translator*. This may seem like an unnecessary complication, when PGPLOT can create the device-specific commands directly, but it has some advantages. Metafiles can be used, for example, to transfer pictures between two computing sites (they are usually smaller than the corresponding device-specific files) or for archiving pictures. It is sometimes convenient to make a program generate a metafile rather than a device-specific file so that one can, for example, take a "quick look" at the picture on an interactive display before making hard copies. One may not know what hard-copy devices will be available when the program runs: for example, if it turns out after your 6-hour batch job has finished that the Versatec printer has broken, it is nice to be able to send the plot to some other device without re-running the batch job.

The metafiles generated by PGPLOT follow the "GSPC Metafile Proposal" described in *Computer Graphics (A. C. M.)*, volume **13**, number 3 (August 1979).

At present, metafiles are available only in the VMS version of PGPLOT, not in the UNIX versions.

8.2 Creating Metafiles

Metafiles may be created using PGPLOT in the same way that any other plot file is created: the device specification consists of a disk file name and a device type `/FILE`, for example `PLOT17.GMF/FILE`. The default file type if none is specified is `.GMF` (for Graphics Meta File). Programs which might generate metafile output should not make any assumptions about the physical scale of the picture; the scale will vary depending on what device the picture is ultimately plotted on.

8.3 Translating Metafiles

In order to generate graphics output from a metafile, one must use a *Metafile Translator* to interpret the device independent metafile commands. The Metafile Translator (GMFPLOT) provided with PGPLOT uses the PG-PLOT subroutines to generate the device-specific output: thus a metafile may be displayed on any of the devices supported by PGPLOT. (A metafile may even be translated into another metafile, but this is not very useful.) To use GMFPLOT, first define a command `PLOT`, say, as follows; this definition may be included in your `LOGIN.COM` file if you make extensive use of it:

```
$ PLOT == "$PGPLOT_DIR:GMFPLOT"
```

The `PLOT` command takes two arguments: the name of the input metafile, and the device specification for the output. The following sample commands display a metafile on a Grinnell and on a Versatec printer:

```
$ PLOT PLOT17.GMF /GR
$ PLOT PLOT17 DEIMOS::LVAO:/VE
```

Again, the default file type for the metafile is `.GMF`.

At present, it is not possible to edit the metafile before display. This is a facility which might be added one day. Nor is it possible to change the scale or *aspect ratio* (ratio of height/width of the display surface). When PGPLOT generates a metafile, it does not know the size or shape of the display surface that it will ultimately be plotted on; as it has to make some assumption, it assumes that the surface will be square. The metafile translator displays the metafile in the largest square available on the output device. Thus a plot which is sent directly to a terminal may not look exactly the same as one that is stored in a metafile and subsequently displayed on the terminal. Future enhancements may allow one to specify the scale and aspect ratio of the metafile when it is generated.