PSgraf

C/C++ Functions for Scientific PostScript Graphics

Version: 3.0.1 June 23, 2004

LEGAL CONSIDERATIONS

PSGRAF is free software; which means that you are free to use it and to redistribute free and verbatim copies of the source code. PSGRAF is not in the public domain, however, it is protected by copyright.

- You are free to modify the source of PSGRAF for your own, private use.
- You may distribute modified, non-commercial versions which retain the name "PSGRAF", provided that (i) you indicate these modifications clearly in the file "PSgraf3.h" under "modifications" as well as at the beginning of every single file that you modified or added, (ii) you distribute it under the same legal terms as PSGRAF is distributed, and (iii) you include the copyright notice and the first paragraph of the file "PSgraf3.h" without any change.
- You may incorporate parts or all of PSGRAF into your own non-commercial software and distribute it, provided that (i) you incorporate the PSGRAF copyright notice at a prominent place in your software and documentation, and (ii) you distribute it under the same legal terms as PSGRAF is distributed.
- You are not allowed to include PSGRAF nor any part of it in any product that is sold commercially, i.e. whose cost exceeds that of media, shipping and handling, without prior written consent of Kurt Roth.

PSGRAF IS LICENSED FREE OF CHARGE. THEREFORE ALL THE FILES WHICH IN THEIR ENTIRETY CONSTITUTE PSGRAF, ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESSED OR IMPLIED. YOU ARE RESPONSIBLE FOR ASCERTAINING THE FITNESS OF PSGRAF FOR ANY SPECIFIC USE, AND CONSEQUENTLY YOU ASSUME ALL THE RESPONSIBILITIES AND COST THAT MAY ARISE FROM USING IT.

Kurt Roth
Institute of Environmental Physics
University of Heidelberg
D-69120 Heidelberg
GERMANY

`kurt.roth@iup.uni-heidelberg.de`
`http://www.iup.uni-heidelberg.de/ts/index.html`

III Appendix 49

1
Preliminaries

1.1 Introduction

PSgraf is a set of C++ functions that facilitates the creation of scientific graphics. Actually, the functions are written in traditional C and C++ is only required for the convenience of operator overloading (see Section 1.6).

The product of PSgraf is a plain ASCII file and as such it may be transferred to and used on many different computers and operating systems. The file contains the device independent PostScript program for rendering the graphics. A number of software products are available that allow PostScript files to be incorporated into text and other graphics, and you may even wish to edit it directly to add some whistles and bells.

A particular strength of PSgraf is its interface to $\TeX$ for labeling the graphics. This allows, for instance, to typeset mathematical expressions and in particular use definitions from the incorporating text.

Notice that PSgraf is a small and rather specialized collection of functions and is certainly not optimal for all and every scientific graphics. Specifically,

1. PSgraf is focused on two-dimensional objects and has only limited support for three-dimensional objects. If you need to show complicated three-dimensional information look for some other tool as for instance IBM's *DataExplorer*.

2. Diagrams and sketches are cumbersome to do with PSgraf and a number of much better tools are available, for example *xfig*. For sketches, scanned hand-drawings are a great alternative.

3. Pixel data – bitmaps and tiffs – can be plotted with PSgraf. However, there are no functions provided for processing such data. Again, excellent tools exist as open source, in particular *QuantIm* and *gimp*.

1.2 Licensing

To allow everybody to tailor PSgraf to his or her particular taste and needs, PSgraf is licensed free of charge (see *Legal Considerations*).

1.3 Installation and Use in a UNIX Environment

The preferable way to use PSgraf is to create a library that can be used by other programs. This may be done with the following steps:

1. Copy the directory `source` from the distribution medium to a convenient place in your system – let this be `$HOME/C/PSgraf3/` – and change the default directory to `source` by
   ```
   cd $HOME/C/PSgraf3/source
   ```

2. Create the library by

   ```
   make
   ```

   which will produce the file `libPSgraf3.a` and move it to directory `/usr/local/lib/`. It will also copy `PSgraf3.h` to `/usr/local/include/`. These locations are suitable for many Unix systems. Modify the last two lines of file `source/makefile` if they are not appropriate for you.

   Notice that writing to directory `/usr/local/` usually requires root privileges. If you do not have them on your system, place the library and the header file into a convenient folder in the search path of your compiler/loader. On many Unix systems this is `Unix/lib/` and `Unix/include/`, respectively. If this does not work you may always specify the locations of the header file and of the library explicitly as explained below.

   Finally notice that the private header `PSgraf_p.h` should not be used by a program that is linked with the library and therefore need not be put into the compiler's search path.

Once the library is created and moved to an appropriate place, programs may use PSgraf functions by including `PSgraf3.h` and by linking the program with the library, e.g., by invoking the compiler by

```
g++ myGraphicsProgram.c [options] -lPSgraf3 [more libs]
```

where [*options*] are compile options and [*more libs*] indicate additional libraries, e.g., `-lm` which is necessary on many systems for linking the math libraries.
If the compiler/loader cannot find the header file or the library, their paths must be provided explicitly. Assuming that `PSgraf3.h` is located in `/a/b/c/include/` and `libPSgraf3.a` in `/a/b/c/lib/`, respectively, the program needs to contain the line

```
#include "/a/b/c/include/PSgraf3.h"
```

and compilation would be invoked by

```
g++ myGraphicsProgram.c [options] -L/a/b/c/lib -lPSgraf3 [more libs]
```

Warning: As an alternative to creating and using a library, you might think of incorporating functions of PSgraf directly into your source code together with the private header `PSgraf_p.h`. This approach is clumsy, however, and is not recommended because of undocumented (private) dependencies between various functions. A particularly bad and insecure programming style would be to directly access or even change information stored in the private part of PSgraf.

1.4 Paradigms

PSgraf uses the metaphor of a drawing paper, a pen, and associated operators. A point on the paper may be defined in *Paper coordinates* (mm on the paper) or in *World coordinates* (user defined scaling). Currently, only Cartesian coordinate systems are implemented. Furthermore, the axes of Paper coordinates and World coordinates are collinear, at least in two-dimensional drawing mode. In a slight generalization, the "drawing paper" may also be three-dimensional in the sense that the pen moves between 3d points either in the up- (just moving) or in the down-mode (drawing a line). This space, whether in Paper coordinates or World coordinates, will be called the *normal space*.
All operations in PSgraf are performed in normal space. If it is three-dimensional, the results are shown as a projection onto an arbitrary plane, the Image Plane. This plane may be linearly distorted in the sense that the coordinate axes need not be orthogonal.
Upon requesting a "new drawing paper" (`gPaper`), the coordinate systems are initialized with default settings such that World coordinates are identical to Paper coordinates. Notice that the settings for Paper coordinates are fixed while those for World coordinates may be changed by `s.WorldCoordinates`. Default values are also assigned to all other parameters in PSgraf such that a reasonable graph can be obtained with a minimal number of commands.

1.5 Data Structures

Notice that PSgraf uses `double` for floating point numbers, not `float`.

1.5.1 Vector in Space

For representing vectors, the data structure `dvec` is provided. It consists of an integer, the dimension, and three doubles for the coordinates of the vector and is defined as

```
struct dvec {
  int dim;              /* dimension */
  double x,y,z;         /* coordinates */
};
```

1.5.2 Color

A color is defined by the color space and by its coordinates in this space.

```
struct color {
  int CS;               /* color space (G=0, RGB=2, HSB=4, CMYK=8) */
  double c0,c1,c2,c3;   /* coordinates */
};
```

1.6 Operator Overloading

PSGRAF adopts the concept of operator overloading from C++ meaning that an operation may be invoked with various parameters but still has the same name. Furthermore, parameters are checked upon compilation. An example of this is the move command which is prototyped as

```
void movea(char CS,struct dvec X);
void movea(char CS,double x,double y);
void movea(char CS,double x,double y,double z);
```

and may thus be invoked in any of the following forms

```
movea('W',X);
movea('W',x,y);
movea('W',x,y,z);
```

where `X` is a two- or three-dimensional vector and `x`, `y`, and `z` are numbers.
In this manual, all the permissable forms of a command are listed together. When overloading is used to allow for explicit specification of vector components, only the first (vector) form is explained. The other forms are then obvious analogs.

1.7 Compatibility with Previous Versions

PSGRAF3 is to a fair degree, but not completely, compatible with PSGRAF2.x. It may thus be prudent to keep the old version operational. Starting with this version, the library created by the default `make` file contains the version number. Hence different versions will not interfere.

1.7.1 Removed Operators

Some operators have been removed because they appeared to be of rather limited use. Following is the complete list of removed operators and possible replacements:

`dRectangle`	replaced by `dParallel`
`initPS`	included into `gPaper`
`sDefault`	fixed defaults set by `gPaper`
`sPaperCoord`	removed without replacement
`sWorldCoord`	removed, use `sXWorldCoord` and `sYWorldCoord`
`WCtoPC`	replaced by `pWP`

1.7.2 Modified Operators

Many operators have been modified to consistently accommodate three-dimensional objects. In most cases compatibility with previous versions was retained through operator overloading. In the following instances, however, the old operator takes rearranged or new arguments or acts in a different way:

`dArrow`	Shape parameters are now set by `sArrowStyle` and the number of arrow types is reduced
`dXAxis`	The first two arguments in the two-dimensional version with explicitly given coordinates are now (x_0, y_0), not (y_0, x_0) as in PSGRAF2.x.
`g.Grid`	Internally, grids are now represented in Paper coordinates. Hence, World coordinates, in 3d also the view, must be defined prior to generating a grid. Similarly, whenever either World coordinates or the view changes, the grid has to be generated anew.

1.8 Usage of this Manual

The manual consists of two parts: the *Reference Manual* where all PSGRAF-operators are described tersely and the *Cook Book* which illustrates their usage for constructing some example graphs.

1.8.1 Reference Manual

The reference manual is intended as a quick introduction to the capabilities and limitations of PSGRAF and as an extended reference for its use. A quick reference for the seasoned user is provided by the header file `PSgraf3.h` which lists all the commands and their allowed arguments.

► Some more advanced aspects of the operators are described in fine print, like this paragraph, and may safely be skipped upon first reading. They generally refer to implementation details, to inherent limitations, or to drawing in three-dimensional space and to the associated complications of coordinate systems and projections.

Many operators can be invoked in different forms that mostly emanate from the different methods to specify a location in space, i.e., vector versus explicit coordinates. Generally, one of these form is the root in that all the others just transform their arguments and then invoke the root operator. In the reference manual, the root operator is indicated by ▼. Using the root operator may be more efficient although the gain is not dramatic for most applications. On the other hand, an alternative form may be more robust against programming errors in that they allow consistency checks. This is in particular the case when vectors are used instead of explicit coordinates.

1.8.2 Cook Book

The cook book is an assortment of complete graphs together with the documented code for their creation and possibly some prose about pertinent issues. It is best used by skimming through the pages, looking at the graphs, and stopping to learn how to produce some interesting feature.

Part I

Reference Manual

2
Two-Dimensional Objects

The main field of application for PSGRAF are two-dimensional graphs which are the focus of this chapter. Three-dimensional operators will be included, however, whenever they fit logically and understanding of the more complicated 3d issues are not important. Hence, scaling of the z-axis is covered together with the scaling of the x- and y-axis but three-dimensional coordinate systems are only introduced in Chapter 3.

2.1 Initialization and Closing

`void` **`gPaper`**`(char*name)`

Provide a new drawing paper and set the graphics state to the default value. This function must be invoked before any other PSGRAF function. The drawing will be saved to the file '`name.eps`' if a new paper is requested or if PSGRAF is ended.

► Upon its first invocation, `gPaper` initializes the internal data structures of PSGRAF. It may be called several times, each time with a different `name` of course, to produce several drawings.

`void` **`endPS`**`(void)`

End PSGRAF, save last drawing paper and release memory. If this function is missing, the last drawing paper will be incomplete and will actually not open in a normal PostScript viewer.

2.2 Coordinate Systems and Clipping

A coordinate system on the drawing paper is required for specifying locations. From PSGRAF's perspective the drawing paper is unlimited. Obviously, this is not true for the PostScript interpreter that eventually renders the graphics. It may thus be desirable to clip the drawing to some finite rectangle.

2.2.1 Coordinate Systems

Only linear Cartesian coordinates are implemented in PSGRAF. There are basically two coordinate systems, Paper coordinates and World coordinates, which refer to the drawing paper and to the world, respectively. In PSGRAF, these systems are collinear and they refer to the "normal space" (Figure 2.1). While the coordinate systems are always collinear, the view – set by `sView` introduced in Chapter 3 – may be employed to rotate the graph.

Paper coordinates are fixed to the drawing paper such that the origin is at the lower left corner, with the x-axis pointing to the right and the y-axis pointing upwards. In three dimensions, the coordinate system is right-handed, the z-axis thus points out of the plane towards the observer. The unit for all Paper coordinates is 1 mm. Drawing tools, in particular the pen's thickness, are defined in Paper coordinates.

World coordinates are used to draw data that are given in a user defined Cartesian coordinate system. This can be shifted and stretched relative to the Paper coordinates, but not rotated. World coordinates are defined by specifying each of the (orthogonal) axes separately by relating two "world points" with the corresponding "paper points". The default setting, which has World coordinates identical to Paper coordinates, is changed for each axis individually by

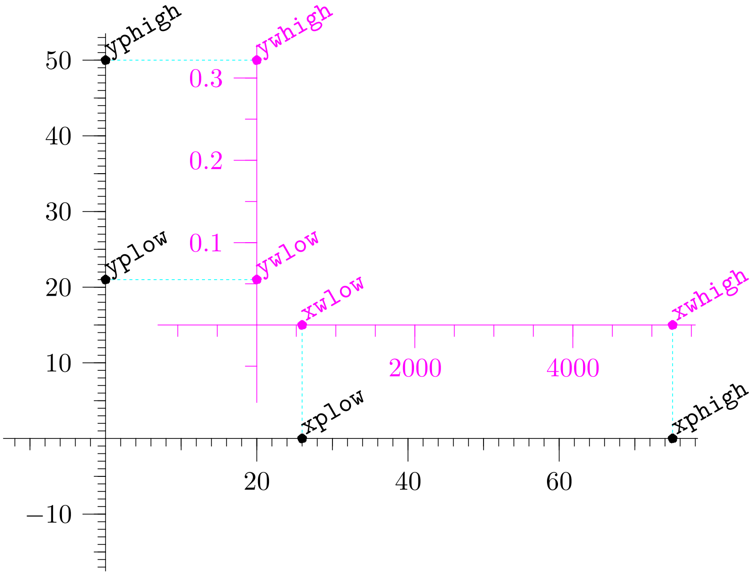


Figure 2.1: PSGRAF knows two different coordinate systems: (i) Paper coordinates (black), which are used to navigate on the drawing paper in much the same way as you would do it using a ruler, and (ii) World coordinates (magenta), which facilitate the drawing of data in their natural coordinate system.

`s.WorldCoord`. This is done by associating two coordinate values, low and high, in World- and in Paper coordinates as illustrated by the dashed cyan lines in Figure 2.1. Notice that the choice of the two values is immaterial, as long as they are different. However, it is often convenient to choose the end points of the corresponding axis in World coordinates. Further notice that the direction of an axis in World coordinates may be changed by relating the value for the high end in World coordinates with the low end in Paper coordinates and vice versa.

`void` **`sXWorldCoord`**`(double xwlow,double xwhigh,double xplow,double xphigh)`

Set the x-component of World coordinates.

`void` **`sYWorldCoord`**`(double ywlow,double ywhigh,double yplow,double yphigh)`

Set the y-component of World coordinates.

`void` **`sZWorldCoord`**`(double zwlow,double zwhigh,double zplow,double zphigh)`
`void` **`sZWorldCoord`**`()`

Set the z-component of World coordinates. Notice that this is only required when working in three dimensions (see Chapter 3) or with rotated World coordinates.

► By default, the z-axis in World coordinates is defined to be identical to the z-axis in Paper coordinates as is the case for the other two axes. However, the default also sets an internal flag that inactivates the third dimension, thereby enabling PSGRAF to better check some arguments. In addition to setting the z-component of World coordinates, `sZWorldCoord` also changes this flag to indicate that the third dimension is active. The default can be reset with `sZWorldCoord()`.

`void` **`pPW`**`(struct dvec Xp,struct dvec*Xw)`
`struct dvec` **`pPW`**`(struct dvec Xp)`
▼ `void` **`pPW`**`(double xp,double yp,double*xw,double*yw)`

Project vector `Xp` in Paper coordinates to `Xw` in World coordinates. With the definitions `struct dvec Xw,Xp`, the projector may be used either as `pPW(Xp,&Xw)` or as `Xw=pPW(Xp)`. Alternatively, Paper coordinates may be given explicitly as (`xp`, `yp`) and the corresponding World coordinates are returned in (`xw`, `yw`).

`void` **`pWP`**`(struct dvec Xw,struct dvec*Xp)`
`struct dvec` **`pWP`**`(struct dvec Xw)`
▼ `void` **`pWP`**`(double xw,double yw,double*xp,double*yp)`

Project vector `Xw` in World coordinates to `Xp` in Paper coordinates. With the definitions `struct dvec Xw,Xp`, the projector may be used either as `pWP(Xw,&Xp)` or as `Xp=pWP(Xw)`. Alternatively, World coordinates may be given explicitly as (`xw`, `yw`) and the corresponding Paper coordinates are returned in (`xp`, `yp`).

▼ `void` **`dXAxis`**`(struct dvec X0,double x1,int below)`
`void` **`dXAxis`**`(double x0,double y0,double x1,int below)`
`void` **`dXAxis`**`(double x0,double y0,double z0,double x1,int below)`

Draw the x-axis in World coordinates from `X0` to `x1`. Notice that `X0` is a vector while `x1` is a simple number. The ticks and numbers are drawn below the axis if `below` is different from

0. With the alternative forms, the origin of the axis is given explicitly in two or in three dimensions. Using a two-dimensional origin in a three-dimensional drawing is an error.

The appearance of the axis is defined by an internal default that may be changed with `sXIntervals` and `sXTicks`.

```
void sXIntervals(double int0,double int1,double int2,int prec)
```

Set the intervals between ticks of orders 0. . .2 on the x-axis. If an interval is 0 or negative, no ticks will be drawn for the corresponding order. If all intervals are 0, the default axis is drawn. Ticks of order 0 are labeled with `prec` decimal places, if `prec` is not negative.

```
void sXTicks(double tick0,double tick1,double tick2)
```

Set the lengths (in Paper coordinates) of the tick marks of orders 0. . .2 on the x-axis.

```
▼void dYAxis(struct dvec X0,double y1,int atleft)
void dYAxis(double x0,double y0,double y1,int atleft)
void dYAxis(double x0,double y0,double z0,double y1,int atleft)
```

Analogous to `dXAxis` but for the y-axis.

```
void sYIntervals(double int0,double int1,double int2,int prec)
```

Analogous to `sXIntervals` but for the y-axis.

```
void sYTicks(double tick0,double tick1,double tick2)
```

Analogous to `sXTicks` but for the y-axis.

```
▼void dZAxis(struct dvec X0,double z1,int below)
void dZAxis(double x0,double y0,double z1,int below)
void dZAxis(double x0,double y0,double z0,double z1,int below)
```

Analogous to `dXAxis` but for the z-axis.

```
void sZIntervals(double int0,double int1,double int2,int prec)
```

Analogous to `sXIntervals` but for the z-axis.

```
void sZTicks(double tick0,double tick1,double tick2)
```

Analogous to `sXTicks` but for the z-axis.

2.2.2 Clipping

A clipping rectangle may be given either in Paper coordinates or in World coordinates. Once invoked, no drawing and no labeling will occur outside this rectangle. Since clipping may be changed arbitrarily, and also be removed, this is a convenient tool for clipping curves or data. See Figure 2.5 for an example.

```
void sClipping(char CS,double xmin,double xmax,double ymin,double ymax)
void sClipping()
```

Restrict drawing to the rectangular area given, in coordinate system `CS`, by the interval [`xmin`,`xmax`] in x- and [`ymin`,`ymax`] in y-direction, respectively. Notice that this command is currently only implemented for two-dimensional graphs.

Clipping is deactivated with `sClipping()` which is also the default setting.

2.3 Color

In PSGRAF a particular color is defined by its coordinates in one of the color spaces G (gray[1]), RGB (red-green-blue), HSB (hue-saturation-brightness), or CMYK (cyan-magenta-yellow-black), which are set by the function `sColorSpace`. Each letter in the name of these spaces denotes a coordinate that can vary in the interval [0,1], where 0 corresponds to minimal intensity of the corresponding property and 1 to maximal intensity.

Example: The color defined by (1,0,0) is red in RGB-space, but black in HSB-space (red with no saturation and no brightness). In HSB-space, the color red is represented by (1,1,1).

```
int sColorSpace(const char*cspace)
```

---

[1] Although G is not a color space in the traditional notation, it is included here to obtain a coherent model.

▼ `int` **`sColorSpace`**`(int cspace)`

Set the color space which will subsequently be used to define a specific color. The function returns 1 if the color space demanded is not defined. Permissible values for the arguments are

`const char`	`int`	color space
`"G"`	0	gray (default)
`"RGB"`	2	red, green, blue
`"HSB"`	4	hue, saturation, brightness
`"CMYK"`	8	cyan, magenta, yellow, black

`int` **`gColorSpace`**`()`

Return the integer code of the current color space. This comes in handy if a particular color space must be used in some function without affecting the external setting.

All functions that accept a gray level, i.e., `sStroke`, `sFill`, or `sContours`, also accept color coordinates. If color is used in conjunction with TeX-typesetting, the corresponding package must be included in the `.tex`-file by adding

`\usepackage{color}`

to the preamble, i.e., before `\begin{document}`.

Examples (the operators `sStroke`, `sFill`, and `sContours` are introduced below):

C functions invoked	action
`sColorSpace("G"); sStroke(1);`	set stroke to black
`sColorSpace("RGB"); sStroke(1,0,0);`	set stroke to red
`sColorSpace("RGB"); sFill(NONE,0,0);`	define empty areas
`sColorSpace("HSB"); sContours(ctr,N,c0,c1,c2);`	set contour attributes

An alternative to setting the color space with `sColorSpace` is the use of color vectors, `struct color`, which set the space implicitly.

2.4 Pens and their Modes

A pen is used to draw a line or curve, but also to fill the resulting area. A line is drawn if the pen's thickness is larger than zero, an area is filled if the fill mode is different from `NONE`.

`void` **`sThickness`**`(double thick)`

Set the thickness, in mm, of the square pen. Set `thick=0`, if you want to draw an object like a symbol or a polygon without a frame.

▼ `void` **`sDash`**`(const char*dpat)`

▼ `void` **`sDash`**`(int ip)`

Set the dash pattern of the pen. Examples: `[] 0` → full line, no dashes; `[2] 0` → two units on, two units off, begin with full dash; `[3 1] 0` → three units on, one units off, begin with dash. The argument after the brackets indicates how many steps in the cycle the pattern shall begin. Check the PostScript reference manual for a more complete description.

Instead of specifying the dash pattern explicitly, one of the predefined patterns shown in Figure 2.2 may be used by giving its number. A negative value for the pattern's number leads to a full line, hence `sDash(-1)` and `sDash("[] 0")` have the same effect. The patterns are defined with respect to the pen's thickness *at the time* `sDash` *is invoked* (Figure 2.3).

`void` **`sStroke`**`(double c0)`

`void` **`sStroke`**`(double c0,double c1,double c2)`

`void` **`sStroke`**`(double c0,double c1,double c2,double c3)`

▼ `void` **`sStroke`**`(struct color c)`

Set the gray value of the pen's color to `c0` $\in [0,1]$. The value 0 corresponds to white, the value 1 to black.

The alternative forms of `sStroke` are used for multidimensional color spaces. The value of `ci` must also be in the interval $[0,1]$. Currently implemented are the three-dimensional spaces RGB and HSB and the four-dimensional space CMYK. Notice that while the pen's thickness does not affect the appearance of letters and numbers, its color does.

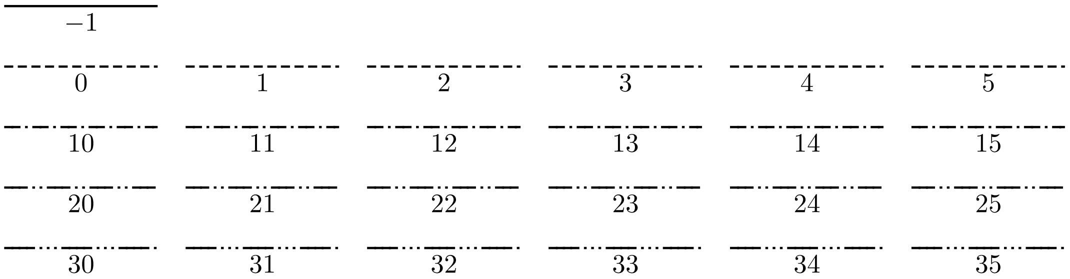


Figure 2.2: The predefined dash pattern `i` is set by `sDash(i)`. The pattern `-1` corresponds to a full line. Any negative value of `i` will actually lead to a full line.

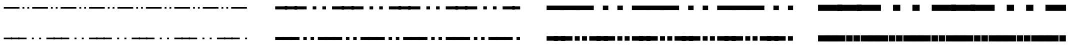


Figure 2.3: Effect of the pen's thickness on pattern 22: For the top row, the pen's thickness was set before the dash pattern while for the bottom row the dash pattern was set before the thickness was changed.

► If the color space in which the stroke is defined is different from the currently active color space, the latter will be changed implicitly and a corresponding warning will be issued. Notice that the intended color space is only defined for the root operator and is guessed for the other ones.

struct color **gStroke**()

Return current pen color.

void **sFill**(double c0)
void **sFill**(double c0,double c1,double c2)
void **sFill**(double c0,double c1,double c2,double c3)
▼void **sFill**(struct color c)
▼void **sFill**()

Set the gray level of the filling pattern to `c0` $\in [0,1]$, where 0 corresponds to white and 1 to black. Every object that has an internal area, like the symbols $0\ldots4$ in Figure 2.6 or a polygon, will be filled with the filling pattern.
For an empty object, set `c0=NONE` or use `sFill()`.
Alternative forms of `sFill` are defined in analogy to `sStroke`.

struct color **gFill**()

Return current fill color.

2.5 Movements and Straight Lines

Movements of the pen and drawing of lines may be described in Paper coordinates or in World coordinates. In each of the following commands, the first argument, CS, is a character (`P` or `W`) that indicates the coordinate system in which the operation has to be performed.

void **movea**(char CS,struct dvec X)
void **movea**(char CS,double x,double y)
void **movea**(char CS,double x,double y,double z)

Move the pen to `(X)`.

void **mover**(char CS,struct dvec X)
void **mover**(char CS,double x,double y)
void **mover**(char CS,double x,double y,double z)

Move the pen by `X` relative to its current position. In general, the current position of the pen is undefined unless a `movea` or `dText` (see below) has been performed previously.

▼void **dLine**(char CS,struct dvec X0,struct dvec X1)
void **dLine**(char CS,double x0,double y0,double x1,double y1)
void **dLine**(char CS,double x0,double y0,double z0,double x1,double y1,double z1)

Draw a line from `X0` to `X1`. The dimensions of the two vectors must be identical.

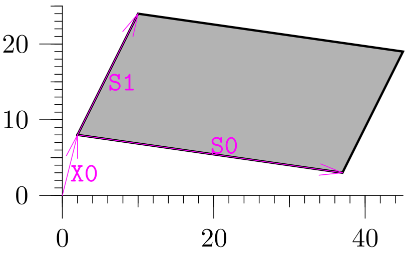


Figure 2.4: Filled parallelogram drawn by `dParallel('P',X0,S1,S2)`.

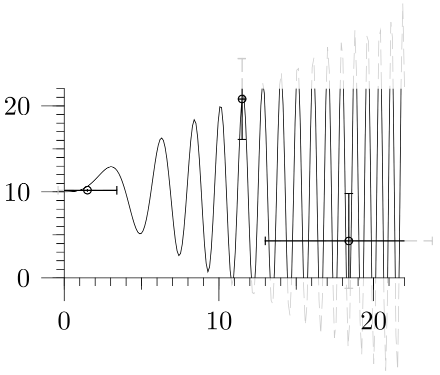


Figure 2.5: Function drawn by `dPolygon` and some data points drawn by `dDataPoint` with default settings. The gray dashed lines represent the unclipped graph while the black lines result from clipping to the rectangle given by the drawn axes.

2.6 Simple Objects

2.6.1 Parallelogram

`void` **`dParallel`**`(char CS,struct dvec X0,struct dvec S0,struct dvec S1)`

Draw a parallelogram with corner `X0` and sides `S0` and `S1` in coordinate system `CS`.

2.6.2 Data Point

`void` **`sDataPoint`**`(int symbol,double rsymbol,double barlength)`

Set the appearance of data points by determining the center `symbol` (see Figure 2.6 for possible values) and its radius [mm] and the width `barlength` [mm] of the end bar. End bars are only drawn if the error bar are larger than the central symbol.

▼ `void` **`dDataPoint`**`(char CS,struct dvec X,struct dvec dX)`
`void` **`dDataPoint`**`(char CS,double x,double dx,double y,double dy)`

Draw data point at location `X` in coordinate system `CS` together with corresponding error bars `dX` which also refer to coordinate system `CS`. The appearence of the data point may be set with `sDataPoint`. In the alternate form, location `(x,y)` and errors `(dx,dy)` are given as components.

2.6.3 Symbol

▼ `void` **`dSymbol`**`(char CS,struct dvec X,double r,int SY)`
`void` **`dSymbol`**`(char CS,double x,double y,double r,int SY)`
`void` **`dSymbol`**`(char CS,double x,double y,double z,double r,int SY)`

Draw the symbol `SY` (see Figure 2.6) at location `X`. It will be contained in a circle of radius `r` (in Paper coordinates).

2.6.4 Arrow

`void` **`sArrow`**`(double lhead,double wtail,double whead,char root)`

Set length `lhead` of arrow head, and widths of tail `wtail` and `whead`, respectively. All lengths are in units of mm. The parameter `root` indicates if `X0` in `dArrow` points to the tail (`'T'`), center (`'C'`), or head (`'H'`) of the arrow. Lower case letters may also be used for this argument.

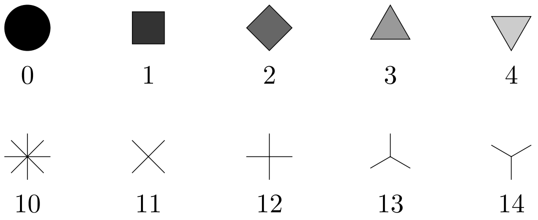


Figure 2.6: Symbol `SY`, contained in a circle of radius `r` centered at `X` in coordinate system `CS`, is drawn by `dSymbol(CS,X,r,SY)`. The symbols on the lower line are created from the ones in the upper line by connecting the corners with the center.

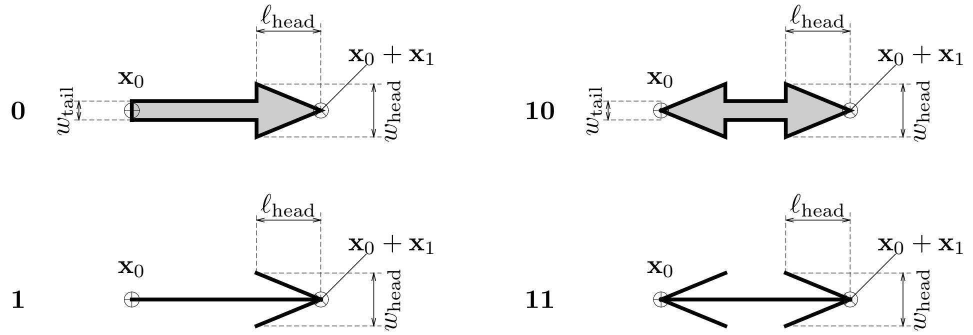


Figure 2.7: Available arrows (`type`) and their specification. The shape of an arrow is set with `sArrow` while type, location and orientation are given in `dArrow`.

void **sAArrow**(double lhead,double wtail,double whead,char root,int tilt)

Analogous to `sArrow` but including the parameter `tilt` that indicates if the annotation should be oriented horizontally on the drawing paper or tilted parallel to the arrow.

void **dArrow**(char CS,int type,struct dvec X1)
▼ void **dArrow**(char CS,int type,struct dvec X0,struct dvec X1)

Draw arrow `type` (Figure 2.7) in coordinate system `CS` starting at `X0`. The arrow is given by `X1` (Figure 2.8). The argument `X0` may be missing if the arrow is to be drawn at the origin. A more general form of `dArrow` is described in Section 3.2.

void **dAArrow**(char CS,int type,struct dvec X1,const char*text)
▼ void **dAArrow**(char CS,int type,struct dvec X0,struct dvec X1,const char*text)

Analogous to `dArrow` but adding annotation `text` to the arrow. Location of the annotation relative to the arrow is according to the current settings of text as set by `sText`.

2.6.5 Polygon

void **dPolygon**(char CS,struct dvec*X,int N)
void **dPolygon**(char CS,double*x,double*y,int N)
void **dPolygon**(char CS,double*x,double*y,double*z,int N)

Draw a polygon through the ordered set $\{\texttt{X}[0], \ldots, \texttt{X}[\texttt{N}-1]\}$ where `X[i]` is a two- or three-dimensional vector. For large polygons, the first form may be inefficient since storage is wasted for the dimension of each point and, if the polygon is two-dimensional, for the third dimension.

► PSGRAF does not impose any limitation on the size of polygons. However, some PostScript interpreters have a rather low limit on the number of points that may be contained in a polygon. A polygon that exceeds the maximum size can neither be printed nor displayed on the screen. Polygons with less than some 1000 points should not encounter this limitation, however.

2.7 Text and Numbers

void **sText**(const char*font,double size,char hadjust,char vadjust)

Set text characteristics. `font` must be the name of a PostScript font (case sensitive) that is currently available on the system, like `"Helvetica"` or `"Helvetica-Oblique"`, or it may be `"TeXfonts"` if the text shall be typeset latter on by TeX.

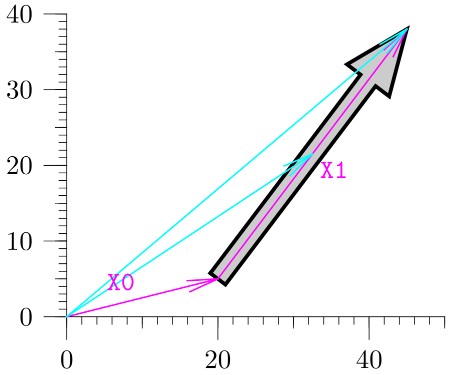


Figure 2.8: Positioning and orientation of an arrow produced with `dArrow('P',0,X0,X1)`. The green arrows indicate `X0` for `root` equal to `'C'` and `'H'`, a parameter set by `sArrow`.

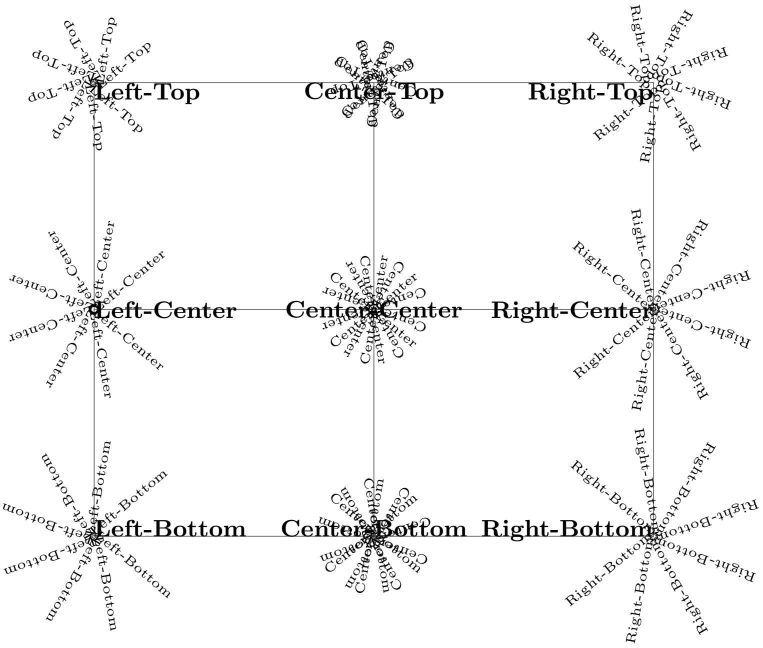


Figure 2.9: Text may be written at any position (x, y), at any angle φ, and at any adjustment relative to (x, y) using `sText`, `sTextRotation`, and `dText`. In this figure, (x, y) are the nodes of the regular grid outlined. The text indicates the adjustment relative to the node.

The size of the font (in pt; 72pt equal 1 inch or 25.4 mm) is given by `size`. It is only effective for PostScript fonts. The size of the TEX fonts are determined when the document is actually typeset.

The horizontal adjustment of the text with the current point is determined by `hadjust`, which must be `'L'`, `'C'`, or `'R'` for **L**eft, **C**enter, and **R**ight. The vertical adjustment is determined by `vadjust`, which must be `'T'`, `'C'`, or `'B'` for **T**op, **C**enter, and **B**ottom.

The baseline of the font is given by the pen's current position and the rotation angle φ (see `sTextRotation` and Figure 2.9).

`void` **`sTeXStyle`**`(const char*texsty,int layered)`

When rendering text with `TeXfonts`, `texsty` is prepended to allow special formatting. This option is needed whenever text is generated automatically by another PSGRAF-command, e.g., `dLegend`. Default for `texsty` is an empty string. For `layered` different from 0, text layering is attempted (see fine print for `dText`. Default is 0, i.e., no layering.

`void` **`sTextRotation`**`(double phi)`

Set the local rotation for the text. The angle `phi` of the text's baseline with the x-axis of the Paper coordinates is measured in degrees (counterclockwise).

`void` **`dText`**`(const char*text)`

Draw `text` at the pen's current position. Unless `movea` or `mover` has been performed as the last PSGRAF function, the current position of the pen is generally not defined, and `dText` will result in an erroneous PostScript file. However, after writing a text, the current position is on the baseline at the end of the text.

▶ Subsequent PSGRAF operators add graphics objects to the drawing that will hide underlying, previously drawn objects. This is also the case with text that may become hidden beneath some filled area, at least if PostScript fonts are used. Such a fine interlayering is not possible with TEX-fonts, however. It is approximated by separating the text into a part that is typeset before the PostScript graphics is rendered, and may thus become hidden, and into a part that is typeset on top of the entire drawing, hence is always visible. The separation is done automatically with all text drawn after the last polygon in the second class.

`void` **`sNumber`**`(char format,int prec)`

Set format for drawing numbers. `format` must be `'f'` for floating point format, `'e'` for exponential format, or `'g'` for mixed format. `prec` is the number of digits after the decimal point.

`void` **`dNumber`**`(double thenumber)`

Draw `thenumber` at the pen's current position in analogy to drawing text with `dText`. Notice that `thenumber` must be a double.

▶ To draw multiple numbers or composites of text and numbers, write everything to a string using `sprintf` which is provided in the standard library `<stdio.h>` and draw this string with `dText`.

2.8 Contour Plots

Contour plots are used for representing two-dimensional scalar data that are defined on some grid. Contours may be drawn as lines and/or by shading. Creating a contour plot requires three steps: (i) definition of the grid, (ii) specification of the contour levels, and (iii) the actual contouring operation.

2.8.1 Grids

All grids are defined in World coordinates. They are essentially two-dimensional, hence can be represented on a possibly distorted plane.

▶ Internally, grids are represented in Paper coordinates. Hence, World coordinates, in 3d also the view, must be defined prior to generating a grid. Similarly, whenever either World coordinates or the view changes, the grid has to be generated anew.

Quadrangular Grids

This class consists of regular square grids and of their distortions. Each internal node thus has exactly four connected neighboring nodes. Grids with a more general topology, e.g., two squares joining the same boundary line of a rectangle cannot be represented as a quadrangular grid and must be described by a triangular grid (described below).

`void` **`gRGrid`**`(struct dvec X0,struct dvec X1,struct dvec X2,int N1,int N2)`
`void` **`gRGrid`**`(double x0,double dx,int Nx,double y0, double dy, int Ny)`

Define a regular grid by its lower left corner `X0` and non-parallel grid vectors `X1`, `X2` with `N1` nodes in 1-direction and `N2` nodes in 2-direction. In the second form, the origin is `(x0, y0)` and the distance between nodes in the x- and y-direction is `dx` and `dy`, respectively. The nodes are numbered starting with 0 for the node at `X0` and increasing in 1 (x) direction (see Figure 2.10).

▶ The distances between nodes may be negative in which case `(x0, y0)` is not the lower left corner any more.

```
void gSRGrid(struct dvec X0,struct dvec X1,double*x1,int N1,
                            struct dvec X2,double*x2,int N2)
void gSRGrid(double*x,int Nx,double*y,int Ny)
```

Define an orthogonal semi-regular grid with corner at `X0`, non-parallel directions `X1` and `X2`, and values `x1` and `x2` in 1- and 2-direction, respectively. Notice that `x1` and `x2` must start with 0 for the first point to coincide with `x0` and that the values must change strictly monotonically (see Figure 2.10).

`void` **`gDRGrid`**`(double*x,double*y,int Nx,int Ny)`

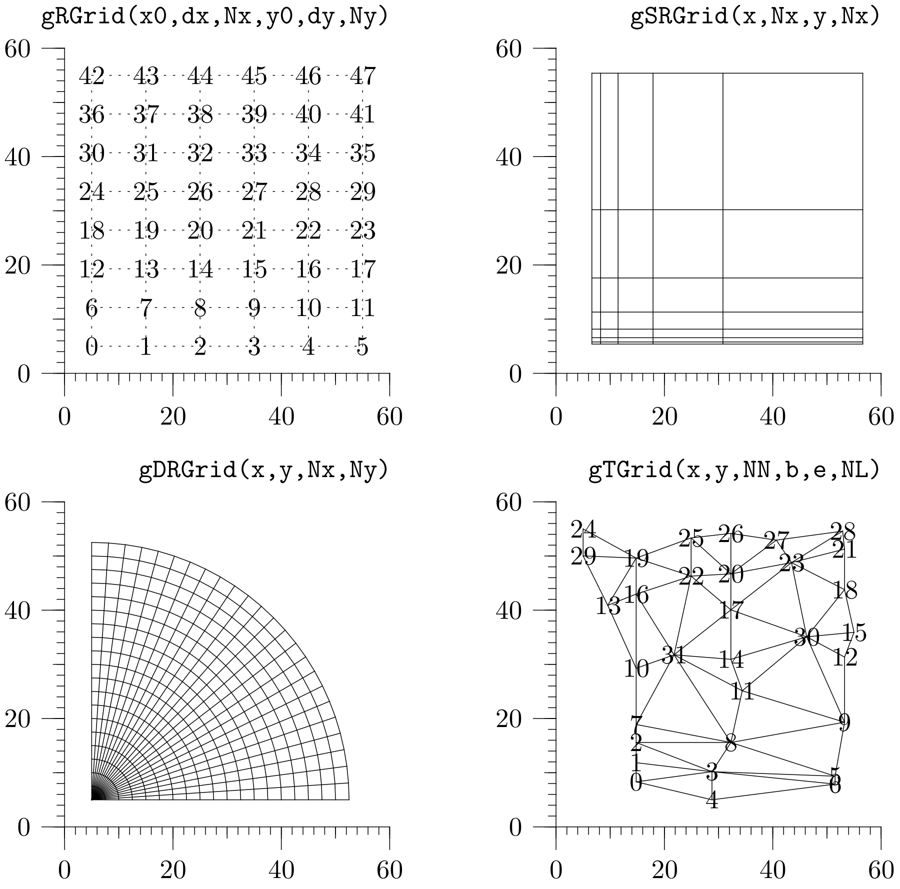


Figure 2.10: Available grids for contouring in PSGRAF are either quadrangular (regular, orthogonal semi-regular, deformed regular) or arbitrarily triangular. The numbers of the grid nodes correspond to the positions of the data in the array supplied to the operator `dContours(...)`.

`void` **`gDRGrid`**`(double*x,double*y,double*z,int Nx,int Ny)`

Define a deformed regular grid which is topologically identical to a regular grid with `Nx` nodes in the x-direction and `Ny` nodes in the y-direction. The coordinates of the `Nx*Ny` nodes are $\{(\mathtt{x[i]},\mathtt{y[i]})|i=0,\ldots,\mathtt{Nx}*\mathtt{Ny}-1\}$.

The contouring operator is not guaranteed to work properly if the coordinates of different grid nodes are identical. However, they may be chosen so near to each other, that they are indistinguishable on any output device (see Figure 2.10 and the corresponding source on page 40).

- The meaning of `Nx` and `Ny` is different from that in `gSRGrid`, which is reflected in the different ordering of the arguments for the two functions. In `gSRGrid` they indicate the size of the arrays `x` and `y`. In contrast, they indicate the structure of the grid in `gDRGrid`; the size of `x` and `y` is identical, namely $\mathtt{Nx}*\mathtt{Ny}$.

Triangular Grids

Many real data set consists of values that have been obtained on a completely irregular grid. Two approaches may be used to represent them: (i) they may be interpolated to a quadrangular grid, e.g., by linear interpolation or kriging, or (ii) they may plotted directly over the irregular grid. The functions in this section facilitate the second approach. Notice that this implies linear interpolation between grid points.

`void` **`gTGrid`**`(double*x,double*y,int NN,int*b,int*e,int NL)`

Define an arbitrarily connected grid that consists of triangular elements (see Figure 2.10). The coordinates of the `NN` grid nodes are given in `x` and `y`. The `NL` grid lines are specified by the numbers of their beginning nodes `b` and ending nodes `e`. They are oriented such that `b[i]` $<$ `e[i]`.

- The implementation of arbitrary triangular grids is not very efficient and only useful for rather small grids for which the number of nodes does not exceeding a few hundred.

- ▶ Only minimal checks for the correctness of the grid definition are implemented. It is recommended to draw the grid with `dGrid(1,1)` after its definition.

`int` **`gTCover`**`()`
Construct the triangular elements defined by the grid lines, calculate their centers of gravity and return the number of elements.

`void` **`iTCover`**`(int NT,int**nodes,double*cgx,double*cgy)`
Inquire the coverage of the grid. For each of the `NT` triangular elements, the ids of its nodes, `nodes`, and the coordinates of the center of gravity, `cgx` and `cgy`, are returned. The dimension of `nodes` is `NT` × 3, that of `cgx` and `cgy` is `NT`. Space for these arrays must be allocated before `iTCover` is called.

Auxiliary Functions

`void` **`dGrid`**`(int Node_numbers,int Line_numbers)`
Draw the current grid with node numbers and line numbers, if the corresponding parameters are equal 1.

`void` **`dGridBoundary`**`(int Node_numbers,int Line_numbers)`
Draw the boundary of the current grid with node numbers and line numbers, if the corresponding parameters are equal 1.

`void` **`deleteGrid`**`(void)`
Delete the currently existing grid and its boundary. Using this function is optional: it is invoked implicitly by `endPS` and whenever a new grid is defined. The latter produces a warning message.

2.8.2 Setting Contour Levels

`void` **`sContours`**`(double*ctr,int N,double*c0)`
`void` **`sContours`**`(double*ctr,int N,double*c0,double*c1,double*c2)`
`void` **`sContours`**`(double*ctr,int N,double*c0,double*c1,double*c2,double*c3)`
`void` **`sContours`**`(double*ctr,int N,struct color*c)`
Define `N` contour lines by their contour values {`ctr[0]`< … <`ctr[N-1]`}. If the function `dContours` (see below) is called with `filled=1`, the region where $z \leq$ `ctr`[i] is filled with the gray level `c0[i]`, where the value 0 corresponds to white and 1 to black. The association between values and gray levels is given in the following table. Notice that the dimension of `c0` is by one larger than that of `ctr`.

value z	gray level
$z \leq$ `ctr`[0]	`c0[0]`
`ctr[i-1]`$< z \leq$ `ctr`[i]	`c0[i]`
`ctr[N-1]`$< z$	`c0[N]`

The alternative versions allow to specify the contour lines and fills as color components in RGB or HSB and CMYK space, respectively, while the last form accepts colors directly.

2.8.3 Contouring

`void` **`dContours`**`(double*Z,int filled,int framed)`
Draw contour lines for the dataset `Z` (see Figure 2.11). Before using this function, the grid (`d..Grid`) and the contour levels (`sContours`) must be defined. The contours are filled with the colors set by `sContours` if `filled` $\neq 0$ and framed with the current stroke color if `framed` $\neq 0$. The parameter `framed` $\in \{-1, 0, 1\}$ determines the contour lines:
-1 : contour lines are only drawn in the interior of the region
0 : no contour lines are drawn (only sensible with `filled`$= 1$)
1 : contour lines are drawn in the interior and on the boundary of the region.
The option `framed` $= -1$ is useful for large and complicated data fields that may produce very long contour lines. As every contour line is internally represented as a polygon of arbitrary length, large data fields tend to generate unprintable PostScript files, see note on page 13. To prevent this from happening, the entire region may be cut into smaller pieces

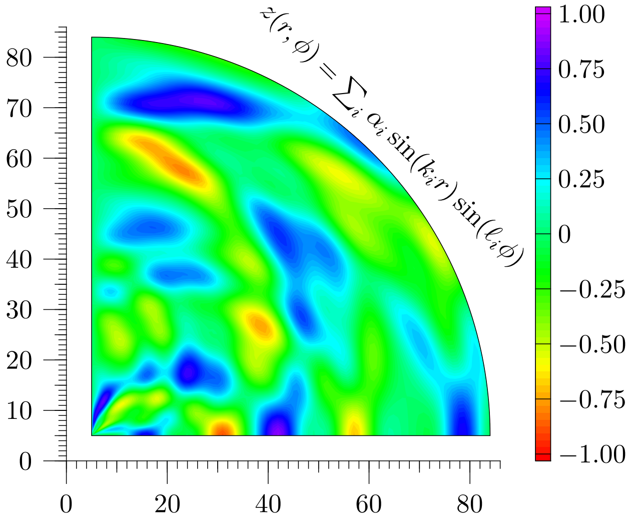


Figure 2.11: Contour lines of a superposition of periodic functions in polar coordinates.

that are then contoured separately. The precision of the PostScript device warrants that the boundaries between adjacent regions are not visible.

`void` **`dLegend`**`(double x0,double y0,double dx,double dy,int horiz,int nth)`

Draw legend for density and contour plots (see Figure 2.11). The lower left corner of the gray bar is at `(x0,y0)` and it extends in x- and y-direction by `dx` and `dy`, respectively. All these parameters refer to Paper coordinates. The orientation of the legend is determined by `horiz` $\in \{0,1\}$. Every |`nth`| contour level is marked by a line in the gray bar and its value is written next to it in the current format for numbers (see `sNumber`). A negative value `nth` indicates that the legend is for a density plot (see `dDensity` below) and that variations should be continuous.

If text rotation is 0 when `dLegend` is called, the numbers are adjusted according to internal default settings. These depend on the orientation of the legend. In particular, the numbers are written below the color bar for a horizontal legend with `dy`> 0 and to the left of the color bar for a vertical legend with `dx`> 0. For `dy`< 0 and `dx`< 0, they are written above and to the right, respectively. If text rotation is different from 0, the current adjustments—set by `sText`—are used. This allows to write the numbers at an angle. If the current adjustment shall be used with horizontal text, `sTextRotation(0.00001)` would do the trick.

Auxiliary Functions

`void` **`sMinPixel`**`(double pixelsize)`

Suppress drawing of polygons in `dContours` which are smaller than `pixelsize` (in PC). The number of suppressed polygons is given as a warning message of `dContours`. Default: `pixelsize` $= 0$, set by `gPaper`.

- Setting `pixelsize` > 0 can drastically reduce the size of complex drawings, e.g., contours of random fields, without compromising the graphical appearance. The caveat is that upon enlarging, such a picture will contain less detail than one with `pixelsize` $= 0$.
- Be aware that setting `pixelsize` to a value which is large than the actual resolution of the output device may lead to inconsistencies along boundaries if the drawing is assembled from separate cuts. They originate from regions which are cut such that one piece is smaller than `pixelsize` while the other one is larger.

`int` **`interpolate`**`(double*Z,int*missing)`

Recursively interpolate values of `Z` that are signaled to be missing by `missing`. In each cycle, missing values are interpolated by averaging their non-missing nearest neighbors. This is repeated until there are no more missing values. The function replaces `Z` and returns the number of interpolated values. It does not affect `missing`, however.

This function is *not* intended to be a sophisticated interpolator: its main use is to facilitate contouring of incomplete datasets. The areas corresponding to missing data should be marked with `dMissing`.

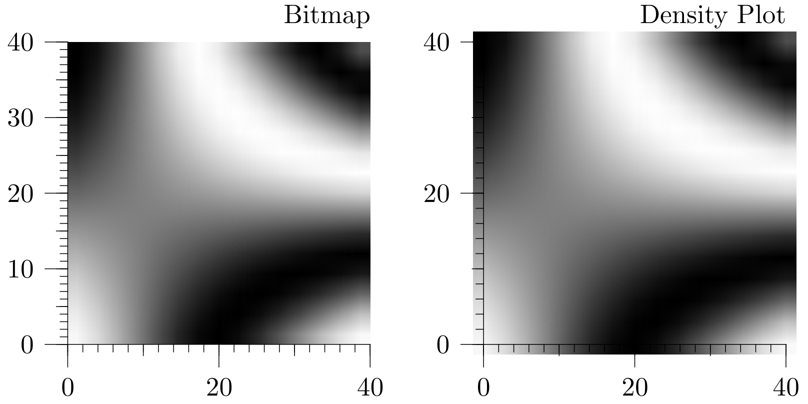


Figure 2.12: Drawing a gray scale bitmap by `dBitMap(0,0,60,60,15,15,gry)`, PSGraf assumes that the rectangle formed by $(0,0)$ and $(60,60)$ in Paper coordinates is to be filled with pixels whose gray values are given in consecutive order in `gry`. In contrast, a density plot is assumed to represent point values, i.e., the point to which the value refers is the center of the pixel. Plotting the data used for the bitmap as a density plot by `dDensity('W',0,0,60,60,15,15,z,1)` therefore produces a larger area.

```
void dMissing(int*missing)
```

Fill areas corresponding to missing data, signaled by `missing`, with the current filling pattern. This function must be called after `dContours`.

2.9 Bitmaps and TIFF Images

In contrast to contour plots, the resolution of bitmaps and TIFF images is inherently limited. This may not affect the final appearance of a plot, however, particularly for fine-grained complex graphics. For these, calculation of a density plot, as opposed to a contour plot, leads to substantially faster performance and much smaller file size.
Bitmaps and TIFF images are currently only available for two-dimensional graphics.

```
void dBitMap(double x0,double y0,double dpx,double dpy,int Nx,int Ny,
           double*gry)
void dBitMap(double x0,double y0,double dpx,double dpy,int Nx,int Ny,
           double*c0,double*c1,double*c2)
void dBitMap(double x0,double y0,double dpx,double dpy,int Nx,int Ny,
           double*c0,double*c1,double*c2,double*c3)
```

Draw bitmap `gry` with origin `(x0,y0)` and distant corner `(x0+dpx,y0+dpy)` in Paper coordinates. For `dpx` > 0 and `dpy` > 0, the origin is at lower left and the distant corner is at upper right. The source of the bitmap contains `Nx` data points in x-direction and `Ny` in y-direction. The gray values of these points are stored in `gry` with the first value referring to the lower left corner and successive values proceeding in x-direction. Values must be in the range $[0 \ldots 1]$. They are transformed to 8 bit values for rendering of the image.
The alternative functions are for the color spaces RGB, HSB, and CMYK, respectively, where `ci` is the bitmap of the corresponding color-component with each of the elements again in [0,1].

```
▼void sDensity(double*val,int N,struct color*c)
void sDensity(double*val,int N,double*c0)
void sDensity(double*val,int N,double*c0,double*c1,double*c2)
void sDensity(double*val,int N,double*c0,double*c1,double*c2,double*c3)
```

Define the relation between value and color for density plots by the `N` pairs $(\texttt{val}[\texttt{i}], \texttt{c}[\texttt{i}])$, where `val` are ordered such that $\{\texttt{val}[0] < \ldots < \texttt{val}[\texttt{N}-1]\}$. Values smaller than `val[0]` are assigned the color `c[0]`, those larger than `val[N-1]` the color `c[N-1]`. Intermediate values are interpolated linearly. The last element of `c`, `c[N]`, is not associated with a value

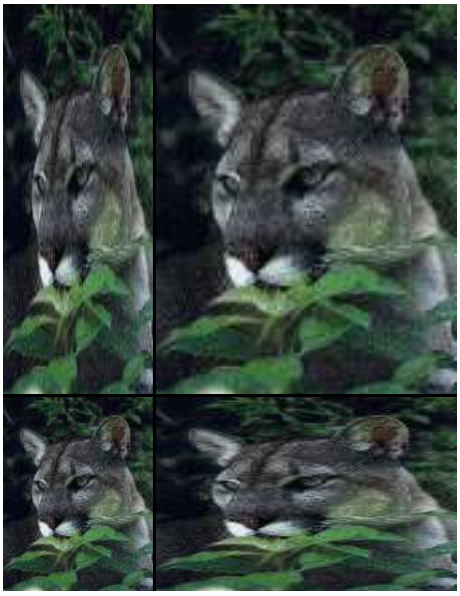


Figure 2.13: The TIFF image `fn.tiff` is drawn by `dTIFF(x0,y0,dpx,dpy,"fn.tiff")` with origin `(x0,y0)` and sizes `dpx` and `dpy`. Choosing any of the sizes 0 maintains the original proportions of the image.

but is used to indicate missing data (see `dDensity`). Notice that the dimension of `val` is `N` while that of `c` is `N+1`.

`void` **`dDensity`**`(char CS,double x0,double y0,double dx,double dy,int Nx,int Ny, double*zd,double misval)`

Draw density function – as a bitmap – of rectangular data `zd` with origin `(x0, y0)` and distant corner `(x0+dx,y0+dy)` in coordinate system `CS`. For `dx` > 0 and `dy` > 0 and natural coordinate system, the origin is at lower left and the distant corner is at upper right. The data array `zd` contains `Nx*Ny` elements which are ordered as

$$\begin{array}{lcl}
\texttt{zd[0]} & \rightarrow & (\texttt{x0},\texttt{y0}) \\
\vdots & & \vdots \\
\texttt{zd[Nx-1]} & \rightarrow & (\texttt{x0}+\texttt{dx}/(\texttt{Nx}-\texttt{1}),\texttt{y0}) \\
\texttt{zd[Nx]} & \rightarrow & (\texttt{x0},\texttt{y0}+\texttt{dy}/(\texttt{Ny}-\texttt{1})) \\
\vdots & & \vdots \\
\texttt{zd[Nx*Ny-1]} & \rightarrow & (\texttt{x0}+\texttt{dx},\texttt{y0}+\texttt{dy})
\end{array}$$

The relation between the values in `zd` and the resulting color is defined by `sDensity` which must be invoked before `dDensity`. Values `zd`[`i`] $>$ `misval` indicate a missing value at location `i` and are mapped to the last color (see `sDensity`).

`void` **`dTIFF`**`(double x0,double y0,double*dpx,double*dpy,const char*fn)`

Draw TIFF image from file `fn` in Paper coordinates with lower left corner `(x0,y0)` and upper right corner `(x0+dpx,y0+dpy)`. The original proportions of the image are maintained if `dpx` $= 0$ or `dpy` $= 0$. For `dpx` $= 0$ image height is `dpy`, for `dpy` $= 0$ image width is `dpx` and for `dpx` $=$ `dpx` $= 0$ natural dimensions are chosen with pixel size 1/72.27 in. The actual values `(dpx,dpy)` used for the drawing are returned.

- In the current implementation, the resulting picture will be 24 bit RBG, independent of the chosen color space and TIFF-format.
- `dTIFF` requires `libtiff` which is available from `http://www.libtiff.org`. It is not part of PSGRAF and is, except for the header files necessary for compilation, not included in this distribution.

3
Three-Dimensional Objects

Two major additions allow the representation of three-dimensional objects in PSGRAF: (i) the z-axis which is defined to point out of the drawing paper and (ii) the arbitrarily positioned and oriented plane onto which the objects are projected. The z-axis is set in complete analogy to the x- or y-axis and the respective operators are described in Section 2.2.1.

Paper coordinates and World coordinates are collinear and they are Cartesian also in the three-dimensional case. A point is thus specified by its coordinates $\{x, y, z\}$.

3.1 Projection

Only two-dimensional projections of the three-dimensional world can be shown on the drawing paper. This plane may be specified either explicitly with `SIPlane` or implicitly by specifying the observer's view with `sView` (Figure 3.1). Notice that only one of these functions need to be executed.

► Distinguish carefully between two-dimensional Paper coordinates and a two-dimensional plane embedded in three-dimensional space at $z = 0$. While the two are identical in normal space, PSGRAF handles them quite differently in that the latter is projected to the Image Plane before being rendered. Depending on the projection, two identical points may thus end up at different locations in the final drawing. If the embedded plane is intended to be represented in two-dimensional Paper coordinates, its third dimension must be removed explicitly.

```
void sView(struct dvec View,struct dvec Right)
void sView(struct dvec X0,struct dvec View,struct dvec Right)
```

Set the view of the 3d object by defining

`X0`	the center at which the observer looks (**0** in the first form)
`View`	the direction from the observer to `X0`
`Right`	the direction that points to the right from the observers perspective

► It is usually desirable to have `View` and `Right` orthogonal. This is not a requirement, however, and PSGRAF will just produce a warning that the drawing will be distorted if the vectors are not orthogonal.

► The vectors `View` and `Right` need not be normalized. The scaling of the graph is determined by `s.WorldCoord` and is not influenced by the magnitudes of `view` and `right`.

```
void SIPlane(struct dvec X1,struct dvec X2)
▼void SIPlane(struct dvec X0,struct dvec X1,struct dvec X2)
```

Set the view of the 3d object by defining a plane onto which the object is projected. This plane is spanned by the vectors `X1` and `X2` and it is positioned such that its origin is at `X0` (**0** in the first form). The vectors `X1` and `X2` also determine the x- and y-direction, respectively, in the image plane.

► In analogy to `sView`, (i) it may be desirable for vectors `X1` and `X2` to be orthogonal and (ii) their magnitude does not influence the scaling of the graph.

```
void pPI(struct dvec Xp,struct dvec*Xi)
struct dvec pPI(struct dvec Xp)
▼void pPI(double xp,double yp,double zp,double*xi,double*yi)
```

Project the vector `Xp` in Paper coordinates onto `Xi` in the two-dimensional Image Plane. With the definitions `struct dvec Xp,Xi`, the projector may be used either as `pPI(Xp,&Xi)` or as `Xi=pPI(Xp)`. As an alternative, the Paper coordinates (`xp`, `yp`, `zp`) may be given explicitely and the corresponding projection is returned as (`xi`, `yi`).

```
void pPW(struct dvec Xp,struct dvec*Xw)
struct dvec pPW(struct dvec Xp)
```

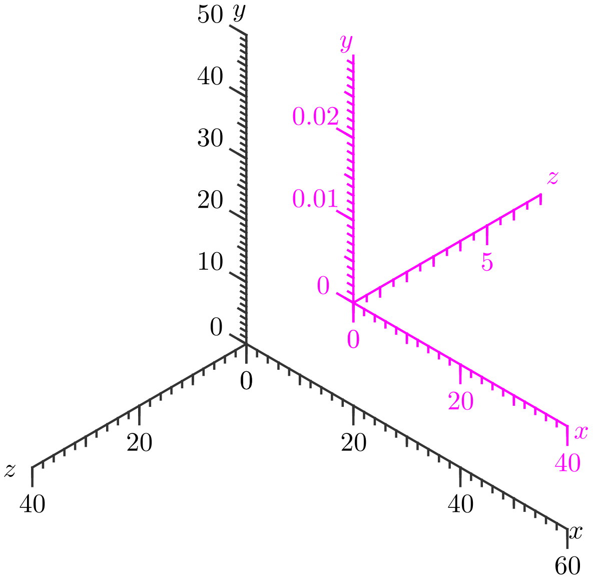


Figure 3.1: Paper coordinates (black) and World coordinates (magenta) with view direction $\mathbf{x}_1 = \{-1, -1, -1\}$ and direction to the right $\mathbf{x}_2 = \{1, 0, -1\}$. A view is set with `sView` or, alternatively, with `SIPlane`.

```
▼ void pPW(double xp,double yp,double zp,double*xw,double*yw,double*zw)
```

Project the three-dimensional vector `Xp` in Paper coordinates into three-dimensional World coordinates, `Xw`. For usage comments, see `pPI`. In the last form, the coordinates are given explicitely.

```
void pWP(struct dvec Xw,struct dvec*Xp)
struct dvec pWP(struct dvec Xw)
▼ void pWP(double xw,double yw,double zw,double*xp,double*yp,double*zp)
```

Project the three-dimensional vector `Xw` in World coordinates into three-dimensional Paper coordinates, `Xp`. For usage comments, see `pPI`. In the last form, the coordinates are given explicitely.

```
void pWI(struct dvec Xw,struct dvec*Xi)
struct dvec pWI(struct dvec Xw)
void pWI(double xw,double yw,double zw,double*xi,double*yi)
```

Project vector `Xw` in World coordinates onto the two-dimensional Image Plane. For usage comments, see `pPI`. There exists no root operator here since `pWI` is implemented by first projecting to Paper coordinates and from there to the image plane.

3.2 Simple Objects

```
void dArrow(char CS,int type,struct dvec X1)
void dArrow(char CS,int type,struct dvec X0,struct dvec X1)
▼ void dArrow(char CS,int type,struct dvec X0,struct dvec X1,struct dvec O)
```

Draw arrow `type` (Figure 2.7) in coordinate system `CS` starting at `X0`. The arrow is given by `X1` (Figure 2.8) and is draw in the plane that contains `O`, that is `O` and `X1` must not be parallel. The argument `X0` may be missing if the arrow is to be drawn at the origin.

```
void dAArrow(char CS,int type,struct dvec X1,const char*text)
void dAArrow(char CS,int type,struct dvec X0,struct dvec X1,const char*text)
▼ void dAArrow(char CS,int type,struct dvec X0,struct dvec X1,struct dvec O,
               const char*text)
```

Analogous to `dArrow` but adding annotation `text` to the arrow. Location of the annotation relative to the arrow is according to the current settings of text as set by `sText`.

```
void dParallel(char CS,struct dvec X0,struct dvec S0,struct dvec S1)
void dParallel(char CS,struct dvec X0,struct dvec S0,struct dvec S1,
                struct dvec S2)
```

Draw a parallelogram with corner `X0` and edges `S0`, `S1`, and `S2` in coordinate system `CS`.

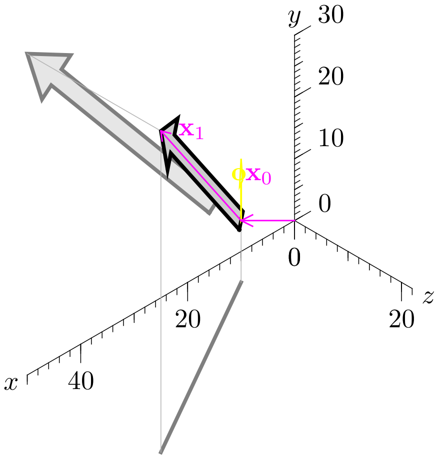


Figure 3.2: Positioning and orientation of an arrow produced with `dArrow('P',x0,x1,o)`, where **o** is parallel to the y-axis. Projections of the arrow onto the xy- and the yz-plane are shown in light gray.

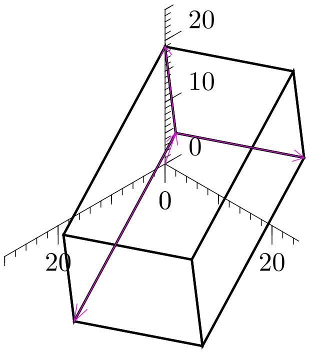


Figure 3.3: Parallelogram drawn in Paper coordinates by `dParallel('P',X0,S1,S2,S3)`.

3.3 Contour Blocks

Consider a three-dimensional, regularly gridded block that is parallel to the World coordinates and contains the data f_{ijk}. Contouring such a block, more precisely its faces, consists of three steps: (i) setting the contour values and fills with `sContours` as described in Section 2.8.2, (ii) describing the data block and the sub-block to be contoured by `sCBlock`, and finally (iii) the actual contouring with `dCBlock` which generates the required grids internally and then invokes `dContours`.

```
void sCBlock(double x0,double dx,int Nx,int il,int iu,
             double y0,double dy,int Ny,int jl,int ju,
             double z0,double dz,int Nz,int kl,int ku,int outer,struct dvec illum)
```

Define the data block and set some additional parameters for contouring blocks. The data block as a whole is defined by its corner {`x0`,`y0`,`z0`}, its grid constants {`dx`,`dy`,`dz`}, and the number of nodes {`Nx`,`Ny`,`Nz`}. The part of the block that is to be contoured is given by the lower and upper node in the three directions. Hence, $\mathtt{il} \le i \le \mathtt{iu}$ and in analogy for the y- and z-components.

For `outer`$= 0$, the inner faces are contoured otherwise the outer ones. In this context "outer" means the face that is encountered first when approaching the block from the viewing direction set by `sView`. These two modes are convenient for creating cut-outs as illustrated in Figure 3.4.

The direction of the illumination is set by `illum`. It is used to dim the color brightness in proportion to the sine of the illumination angle. If the dimension of the vector `illum` is different from 3, the effect of illumination is disabled.

▶ Contour values and contour fills are set with `sContours` as in the case of two-dimensional contours.

```
void dCBlock(double***f,int filled,int framed)
```

Contour the block `f` where, in contrast to `dContours`, the data are contained in a three-dimensional array. The parameters `filled` and `framed` determine the appearance of the contours.

▶ The contouring is done using the two-dimensional operator `dContours`. The required grids are generated automatically which forces preexisting user-defined grids to be deleted.

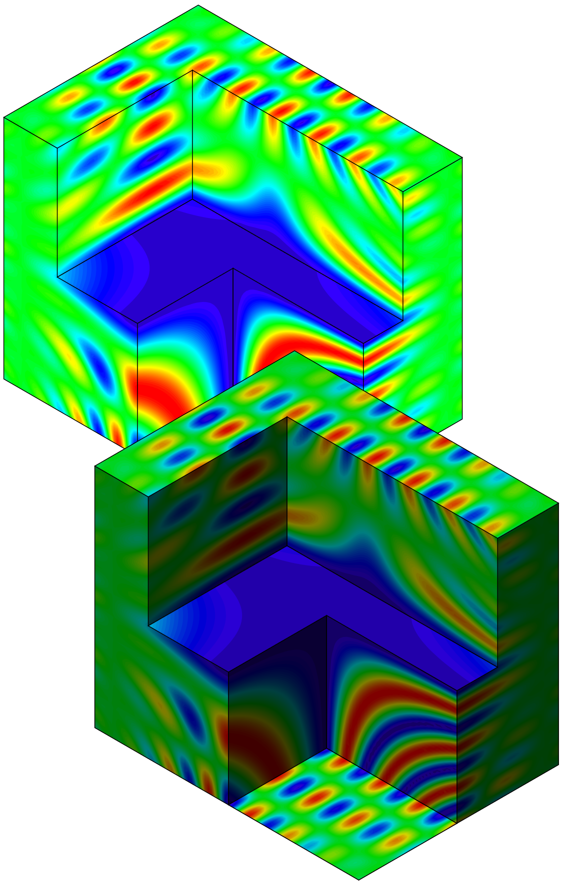


Figure 3.4: Three-dimensional dataset illustrated by first drawing an outer contour block and then two smaller inner ones. The object in the back was drawn with diffuse illumination while the one in front is illuminated by a directed light source overhead.

4

Auxiliary Functions

4.1 Output Options

`void `**`sPreview`**`(int preview)`

Specify if bitmap preview is to be encapsulated (`preview`=1) or not (`preview`=0, default). Including the preview allows displaying the figure even if PostScript rendering is disabled or otherwise unavailable.

- The preview is generated by `gPreview` which uses system calls and thus makes the program system-dependent. Adapting PSGraf to another operating system generally demands the modification of `gPreview`. An alternative is to invoke it manually, on many Unix systems by

  ```
  ps2epsi fn.eps fn.epsi
  ```

  where `fn` is the name of the PostScript file.

4.2 Arithmetic Operations

`void `**`MinMax`**`(double*X,int N,double*xmin,double*xmax)`

Find the minimum `xmin` and the maximum `xmax` in the array `X` of dimension `N`.

4.3 Vector Operations

`struct dvec `**`identity`**`(struct dvec x)`

Return `x`.

`struct dvec `**`negvec`**`(struct dvec x)`

Return −`x`.

`double `**`norm`**`(struct dvec x)`

Norm of vector `x`, i.e., $|\mathbf{x}|$.

`void `**`normalize`**`(struct dvec*x)`
`struct dvec `**`normalize`**`(struct dvec x)`

Normalize vector `x`, i.e., $\mathbf{x}/|\mathbf{x}|$. In the first version the original vector is replaced by the normalized vector, whereas the second form leaves `x` untouched.

`struct dvec `**`addvec`**`(struct dvec a,struct dvec b)`

Add two vectors $\mathbf{c} = \mathbf{a} + \mathbf{b}$ and return $\mathbf{c}$.

`struct dvec `**`subvec`**`(struct dvec a,struct dvec b)`

Add two vectors $\mathbf{c} = \mathbf{a} - \mathbf{b}$ and return $\mathbf{c}$.

`struct dvec `**`scalmult`**`(double a,struct dvec b)`

Multiply scalar with vector $\mathbf{c} = a\mathbf{b}$ and return $\mathbf{c}$.

`double`**`dotprod`**`(struct dvec a,struct dvec b)`

Dot (scalar) product $c = \mathbf{a} \cdot \mathbf{b}$.

`struct dvec `**`vecprod`**`(struct dvec a,struct dvec b)`
`void `**`vecprod`**`(struct dvec a,struct dvec b,struct dvec*c)`

Vector product $\mathbf{c} = \mathbf{a} \times \mathbf{b}$. Notice that `c` is a three-dimensional vector, also if `a` and `b` are two-dimensional.

5
Private Functions

As the title indicates, the functions documented in this chapter are not intended for use in a normal PSGRAF program. They are, and should only be, used as low-level building blocks in higher-level PSGRAF functions.

PSGRAF attempts to check as far as possible the arguments of all commands for consistency. For instance, it checks the coordinate system and the dimension for each operator. While this is convenient in that typos and context errors are signaled, it is also inefficient because the same test may be carried out several times at different levels. To prevent this, some low-level functions are provided that do not perform any checks but presume that this has been done at the higher level.

In order to emphasize the private nature of the low-level functions they are prototyped in `PSgraf_p.h` not in `PSgraf3.h`.

void **l2Pa**(double x,double y)
: Draw a line from the current position of the pen to absolute location $(\mathtt{x}, \mathtt{y})$ in two-dimensional Paper coordinates.

void **l2Wa**(double x,double y)
: Draw a line from the current position of the pen to absolute location $(\mathtt{x}, \mathtt{y})$ in two-dimensional World coordinates.

void **m2Pa**(double x,double y)
: Move to absolute location $(\mathtt{x}, \mathtt{y})$ in two-dimensional Paper coordinates.

void **m2Wa**(double x,double y)
: Move to absolute location $(\mathtt{x}, \mathtt{y})$ in two-dimensional World coordinates.

void **m3Pa**(double x,double y,double z)
: Move to absolute location $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ in three-dimensional Paper coordinates.

void **m3Wa**(double x,double y,double z)
: Move to absolute location $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ in three-dimensional World coordinates.

▼ void **dP2P**(double*x,double*y,int N)
: Draw polygon $(\mathtt{x}, \mathtt{y})$ consisting of $\mathtt{N} > 2$ points in two-dimensional Paper coordinates.

void **dP2W**(double*x,double*y,int N)
: Draw polygon $(\mathtt{x}, \mathtt{y})$ consisting of $\mathtt{N} > 2$ points in two-dimensional World coordinates.

void **dP3P**(double*x,double*y,double*z,int N)
: Draw polygon $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ consisting of $\mathtt{N} > 2$ points in three-dimensional Paper coordinates.

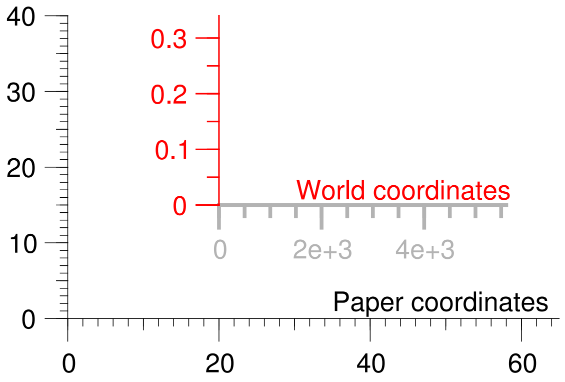
void **dP3W**(double*x,double*y,double*z,int N)
: Draw polygon $(\mathtt{x}, \mathtt{y}, \mathtt{z})$ consisting of $\mathtt{N} > 2$ points in three-dimensional World coordinates.

Part II

Cook Book

This part contains examples of increasing complexity with the intention of providing (i) a more extensive introduction and discussion of PSGRAF-operators and (ii) quick solutions to common drawing problems. The complete code of the examples is included in the PSGRAF distribution in directory `examples`.

Coordinate Systems and Pen Attributes



The code `examples/coordinates.cpp` sets up the drawing paper, draws some axes in Paper coordinates and in World coordinates, and annotes them. In order to illustrate the overall structure of a PSGRAF program, the code is reproduced in its entirety in the following.

```
#include <PSgraf3.h>
```

PSGRAF functions become available by including the underlaying definitions `PSgraf3.h`. As mentioned earlier, refrain from including the private definition of PSGRAF except, maybe, if you are developing new PSGRAF functions.

```
int main()
{
  gPaper("fig/coordinates");
```

The first PSGRAF command must be the request for a drawing paper. This also specifies the file for the PostScript output. Here, output will be written to file `coordinates.eps` that will be stored in directory `fig`. Notice that the extension is added automatically. Obviously, you must have write-permission for this location.

```
      dXAxis(0,0,65,1); dYAxis(0,0,40,1);
```

Draw the x-axis at $y = 0$ mm from $x = 0$ mm to $x = 65$ mm and label it below. Similarly, draw the y-axis at $x = 0$ mm and label it at the left. Notice that axes are always drawn in World coordinates. Since these have not been defined explicity yet, they are, by default, identical to Paper coordinates. Hence all units are in mm. Also the pen is in its default state with lines drawn in solid black with a thickness of 0.1 mm.

```
      movea('P',35,1); dText("PaperCoordinates");
```

Move to location $(x, y) = (35, 1)$ in Paper coordinates and write the string `Paper coordinates`. No font has been specified yet, hence the text is written with the default settings: 10 point Helvetica PostScript font with the bottom-left alignement of the text string with the current location.

```
      sXWorldCoord(0,5600,20,58); sYWorldCoord(0,0.34,15,40);
      sXIntervals(2000,500,0,0);  sYIntervals(0.1,0.05,0,1);
```

Define World coordinates in x-direction by associating $x_w = 0$ with $x_p = 20$ mm and $x_w = 5600$ with $x_p = 58$ mm, where the subscripts w and p indicate World coordinates and Paper coordinates, respectively. Similarly, World coordinates in y-direction are defined. Then, intervals for ticks on the x-axis are set to 2000 for the 0-order, i.e., those that are labeled. First-order ticks are set 500 intervals and second-order ticks are suppressed. The forth argument specifies that labeling shall be done with zero decimal digits which also suppresses the decimal point. Similarly, intervals for the ticks on the y-axis are set and labeling with one decimal is requested.

```
    sStroke(0.3); sThickness(0.5); dXAxis(0,0,5600,1);
```

Set the pen’s color, the stroke, to 0.3 and its thickness to 0.5 mm. The stroke refers to the default color space (G) and thus leads to gray lines with 50% saturation. With this draw the x-axis at $y = 0$, from $x = 0$ to $x = 5600$, and label it below. Notice that (i) the interval over which the axis is draw may be different from the interval used in defining World coordinates with `sXWorldCoord` above and (ii) the pen’s stroke also applies to drawing text.

```
    sColorSpace("RGB");
    sStroke(1,0,0); sThickness(0.2); dYAxis(0,0,0.34,1);
```

Request the RGB color space, a red pen with thickness 0.2 mm, and draw the y-axis.

```
    movea(’W’,1500,0.01); dText("WorldCoordinates");
```

Move to location $(1500, 0.01)$ in World coordinates and write the string `World coordinates`.

```
  endPS();
  return 0;
}
```

Finally close PSgraf and exit the main program. Notice that the drawing will be incomplete if `endPS` is not invoked. The resulting file actually cannot be opened by a regular PostScript interpreter.

On a Linux system, this example may be compiled and executed by first changing into the directory that contains the source `coordinates.cpp` and then typing

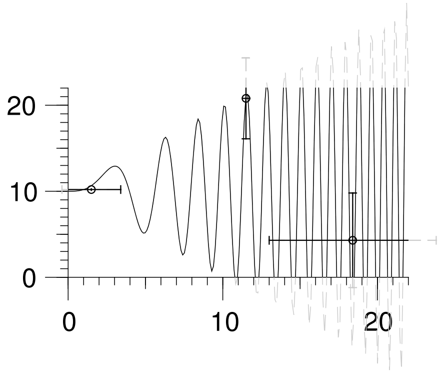
```
g++ -o _coordinates coordinates.cpp -lPSgraf3
_coordinates
```

Notice that in order to execute `_coordinates` the directory that contains `coordinates.cpp` must also contain the directory `fig` to which you have write-permission.
Finally, the resulting file may then be viewed by either typing

```
gv fig/coordinates.eps
```

or, on most systems, by simply clicking on it.

Data Points and Functions



The code `examples/data-function.cpp` illustrates the use of the data structure `dvec`, plotting of data points with error bars, plotting of functions, and the use of clipping.

```
#include <PSgraf3.h>
#include <math.h>
```

In addition to `PSgraf3.h`, we also need to include `math.h` since some mathematical functions are used be the code. On most systems, it further necessary to invoke the compiler with `-lm` to load the mathematics library.

```
#define N       200
```

Define the constant `N` = 200 for the number of points that are going to be used to represent a function. Notice that PSGRAF cannot really plot a function but rather connects a series of points with straight lines. For non-pathological functions, this leads to an arbitrarily good representation.

```
int main()
{
  double x[N],y[N];
  struct dvec X[3],dX[3];
```

Define the arrays `x` and `y` of double precision numbers that are going to contain the coordinates of the series of points. Recall that PSGRAF operates with double precision numbers. Further define the two arrays `X` and `dX` of structures `dvec` that will contain the coordinates of some data points and their errors.

```
  for (i=0;i<N;i++) {
    x[i]=i*22./(N-1);
    y[i]=10.+x[i]*sin(0.2*pow(x[i],2));
  }
```

For points with equal distance in the interval $[0, 22]$ in x-direction calculate points on the graph of $10 + x\sin(0.2x^2)$.

```
  for (i=0;i<3;i++) {X[i].dim=dX[i].dim=2;}
  X[0].x= 1.5; dX[0].x=1.9; X[0].y=10.2; dX[0].y=0.1;
  X[1].x=11.5; dX[1].x=0.2; X[1].y=20.8; dX[1].y=4.7;
  X[2].x=18.4; dX[2].x=5.4; X[2].y= 4.3; dX[2].y=5.5;
```

Set the dimensions of `X` and `dX` and assign some values.

```
    sXWorldCoord(0,22,0,45); sYWorldCoord(0,22,0,25);
    dXAxis(0,0,22,1);        dYAxis(0,0,22,1);
    sStroke(0.2); sDash("[2 1] 0");
    dPolygon('W',x,y,N);
```

Set World coordinates, draw the x- and y-axis, and set the pen's stroke and dash pattern (explained later). Finally draw the polygon defined by the `N` points (`x`,`y`) in World coordinates.

```
sThickness(0.15);
for (i=0;i<3;i++) dDataPoint('W',X[i],dX[i]);
sThickness(0.1);
```

Set the pen's thickness to 0.15 mm, draw the three data points `X` with error bars given by `dX`, and reset the pen's thickness. Notice that error bars are only drawn if the are larger than the symbol that represents the data point. If you wish to change the appearance of the data points, invoke `sDataPoint`.

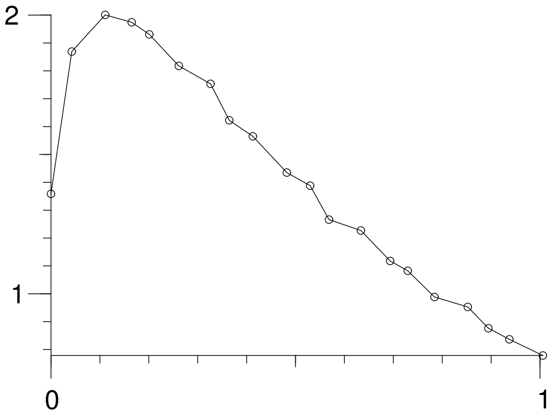
```
sClipping('W',0,22,0,22);
```

Set the clipping window to extend, in World coordinates, in both x- and y-direction from 0 to 22. After a clipping window has been invoked, no drawing beyond its boundaries is possible anymore. This is handy to represent data or functions that contain some spikes which shall be cut. Notice that a clipping window may be defined either in Paper coordinates or in World coordinates but that it will be active in both coordinate systems. To stop clipping again, invoke `sClipping()`.

```
sStroke(1); sDash("[] 0");
dPolygon('W',x,y,N);

sThickness(0.15);
for (i=0;i<3;i++) dDataPoint('W',X[i],dX[i]);
```

Reset the pen's stroke to black and the dash pattern to solid line and re-plot the now clipped data.

Datafile



```
20
0.000000   1.358652
0.042512   1.869169
0.110832   2.000719
0.164950   1.974227
0.201074   1.930818
0.261304   1.817872
0.326261   1.753496
0.364512   1.622678
0.412732   1.565096
0.482173   1.435020
0.529965   1.388011
0.568451   1.266149
0.633707   1.227134
0.693536   1.117744
0.729582   1.082477
0.784143   0.988516
0.852298   0.952764
0.894458   0.876252
0.937329   0.836560
1.005804   0.778536
```

The code `examples/data.cpp` illustrates reading of data from some data file and plotting them.

```
#include <PSgraf3.h>
#include <stdio.h>
#include <stdlib.h>
```

We need to include `stdio.h` for handling files and `stdlib.h` to allocate memory dynamically.

```
double *x,*y,minx,maxx,miny,maxy;
FILE *dat;
```

Define two pointers, `*x` and `*y`, to arrays of type double which will eventually hold the data and variables for storing their minimum and maximum values. Define a pointer, `*dat`, to a file structure.

```
dat=fopen("data/ran0.dat","r");
if (dat==NULL) {
  fprintf(stderr,"cannot open file: abort\n");
  exit(EXIT_FAILURE);
}
```

Open file `data/ran0.dat` for reading and point `dat` to it. If the file cannot be opened for any reason, `fopen` will return `NULL`. This is used here to ascertain successful opening. Notice that such assertions may appear a bit clumsy – the program would work without them – they may save a lot of debugging time and are almost always worth the additional effort.

```
fscanf(dat,"%i",&N);
```

Read the first entry of the data file; the number of entries. Notice that there are ways for reading data without knowing their number in advance. However, these are typically a bit more difficult to program than what we do here.

```
x=(double*)malloc(N*sizeof(double));
if (x==NULL) {
  fprintf(stderr,"cannot allocate memory for array x with %i elements: abort\n",N);
  exit(EXIT_FAILURE);
}
y=(double*)malloc(N*sizeof(double));
if (y==NULL) {
  fprintf(stderr,"cannot allocate memory for array y with %i elements: abort\n",N);
  exit(EXIT_FAILURE);
}
```

Allocate memory for storing the data, again ascertaining that the operation was successful.

```
for (i=0;i<N;i++) fscanf(dat,"%lf %lf",x+i,y+i);
fclose(dat);
```

Read the data from the file into the arrays `x` and `y` and then close the data file. Compare the two `fscanf` commands in this program to learn the differences in reading a single number and an element of some array.

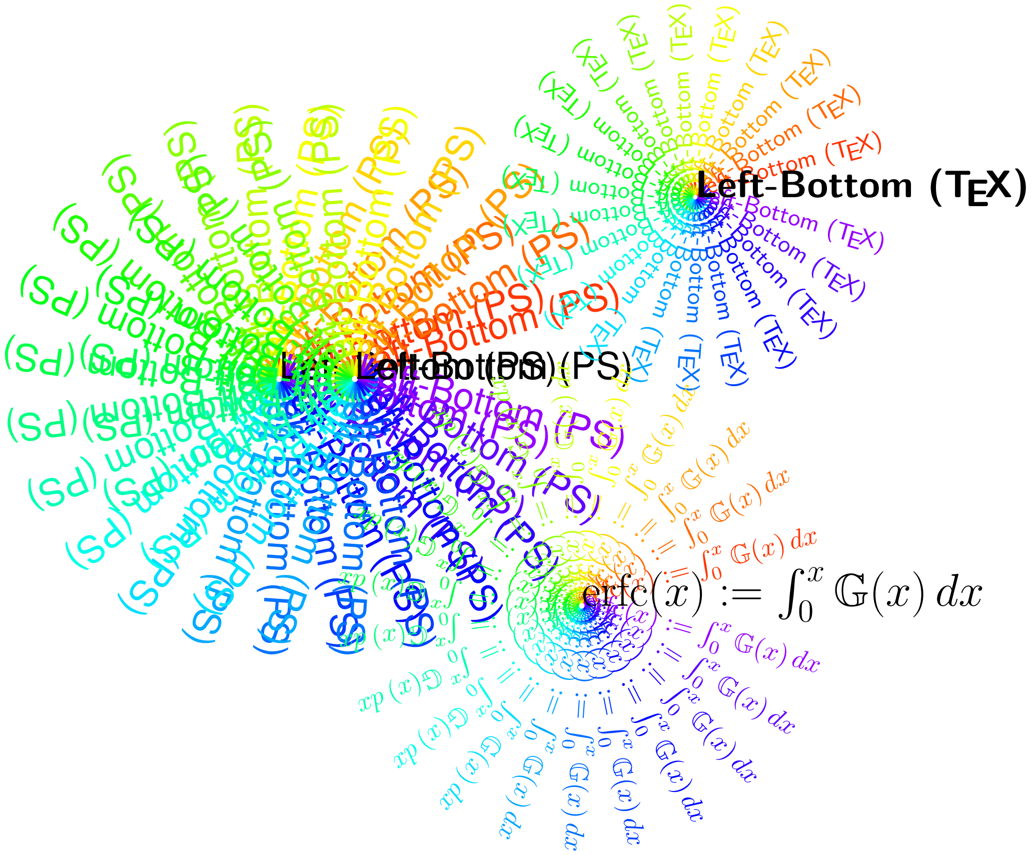
```
gPaper("fig/data");
  MinMax(x,N,&minx,&maxx); MinMax(y,N,&miny,&maxy);

  sXWorldCoord(minx,maxx,0,45); sYWorldCoord(miny,maxy,0,25);
  dXAxis(minx,miny,maxx,1);     dYAxis(minx,miny,maxy,1);
```

Get a paper, determine minimum and maximum values of the two arrays, scale World coordinates accordingly, and draw the axes.

```
  for (i=0;i<N;i++) dSymbol('W',x[i],y[i],0.5,0);
  dPolygon('W',x,y,N);
```

Draw a symbol for each data point and finally connect them with a polygon.

Text



The code `examples/text.cpp` draws some text at various angles and with various colors using both PostScript- and TEX-fonts.

```
double a,da=360./23;
char cbuf[127],*s="$\\mathrm{erfc}(x):=\\int_0^x\\mathbb{G}(x)\\,dx$";
```

Define variables for rotation angle, its increment, a character buffer, and some text string to draw.

```
sColorSpace("HSB");
```

Set color space HSB (hue, saturation, brightness).

```
sText("Helvetica",14,'L','B');
```

Request PSGRAF to use the PostScript font `Helvetica` at 14 points and to adjust it such that the left (`'L'`) bottom (`'B'`) of the text string is aligned with the current location of the pen. Instead of Helvetica, you may use the name of any PostScript font that is available on the system on which the final drawing is rendered. Notice that the alignment arguments are not case-sensitive but that the name of the font is.

```
movea('P',0,0);
for (a=da;a<360;a+=da) {
  sTextRotation(a); sStroke(0.8*a/360,1,1); dText("Left-Bottom (PS)");
}
```

Move the current location to (0,0) and draw the text `Left-Bottom` rotated by incremental angle `a` – in degrees counter-clock-wise – and with incremental hue. Hue may vary between 0 and 1. However, since the end points produce the same color the range is restricted to the interval [0,0.8].

```
sTextRotation(0); sStroke(0,1,0); dText("Left-Bottom (PS)");
```

Reset text rotation to 0 and the stroke to black – any hue and saturation but no brightness – and draw the string again.

```
sText("TeXfonts",14,'L','B');
movea('P',80,18);
for (a=da;a<360;a+=da) {
  sTextRotation(a); sStroke(0.8*a/360,1,1); dText("{\\small\\sf Left-Bottom (\\TeX)}");
}
sTextRotation(0); sStroke(0,1,0); dText("{\\Large\\bsf Left-Bottom (\\TeX)}");
```

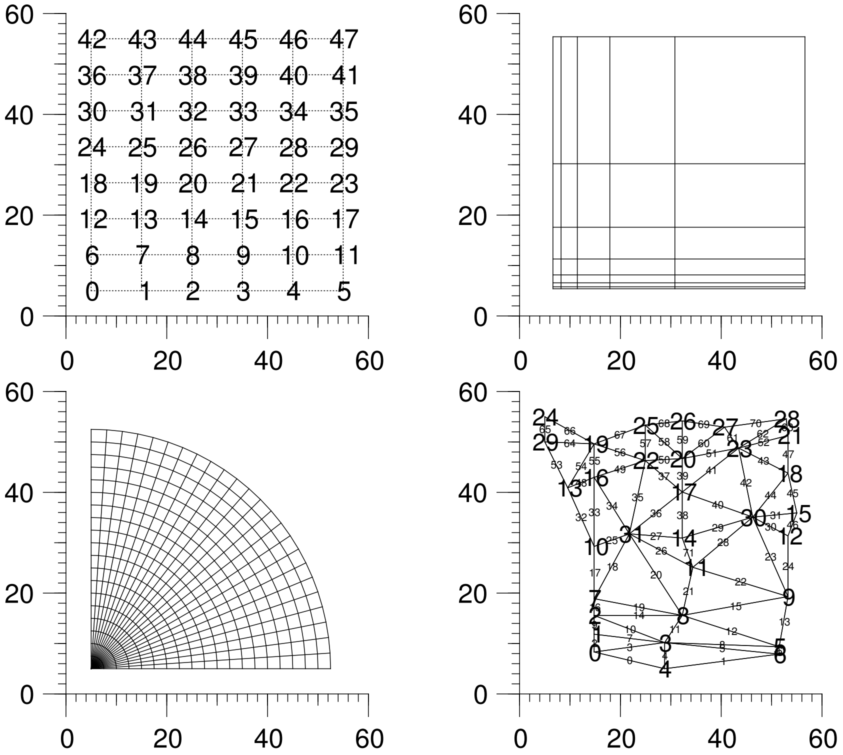
Request PSGRAF to use TEX-fonts instead of PostScript fonts. The other text settings remain unchanged. Whenever TEX-fonts are used, labeling commands are collected into a separate file that will eventually be processed by TEX's typesetting engine. This file will be stored with the same name as the drawing but with extention `tex`. As a consequence of this two-step procedure, the PostScript file produced by PSGRAF is incomplete.
A further consequence of using TEX-fonts is that the font size is *not* determined by the second argument of `sText`, but by the font size of the TEX-document at the moment when the graphics is typeset. This is illustrated in this example: the PostScript font in the drawing is actually 14 point, while the TEX-font is much smaller, namely 10 point. Nevertheless, it is good practice to specify the correct font size with `sText` since this information is used by some operators that do automatic text adjustment, e.g., when labeling the axes.
The main advantage of using TEX-fonts is, besides the aesthetics of a uniform font across the entire document, the ability to use special characters, e.g., `\\TeX` in the text string, and in particular symbols that are private to the document. The latter is for instance the case with the TEX-macro `\bsf` that has been defined to produce sans-serif bold text for this manual. (Notice that to produce the backslash \ in C or C++, a double-backslash must be typed.)

```
movea('P',55,-30);
for (a=da;a<360;a+=da) {
  sTextRotation(a); sStroke(0.8*a/360,1,1); dText(s);
}
sprintf(cbuf,"{\\LARGE%s}",s);
sTextRotation(0); sStroke(0,1,0); dText(cbuf);
```

Instead of giving the text directly as argument to `sText`, a string like `s` may be given or it may be convenient to first accumulate text, and possible numbers, into some character buffer like `cbuf` which then drawn by `sText`. Finally notice that you may arbitrarily switch between PostScript and TEX-fonts.

Special TEX-macros are required to for labeling PSGRAF drawings. An example style file that does this job is included in the PSGRAF distribution as `PSTeX/PSgraf.sty` and is also shown in Appendix A.

Grids



The code `examples/grids.cpp` creates and draws examples of the different grids available in PSGRAF.

```
int Nx=25,Ny=20,Np,Nl,*b,*e,i,j;
double x[Nx*Ny],y[Nx*Ny],sphi,cphi,dphi=1.5707963/(Nx-1),r,dr=2.5,min,max;
struct dvec orig;
FILE *f;

orig.dim=2; orig.x=0; orig.y=0;
```

Define the variables to be used, initialize them where appropriate, and define the two-dimensional vector `orig` to the origin.

```
gRGrid(5,10,6,5,7.1429,8);
```

After getting a paper, initializing World coordinates, and drawing the axes, get a regular grid with origin (5,5), 6 *nodes* in x-direction separated by the grid constant 10, and 8 nodes in y-direction with grid constant 7.1429.

```
sDash(0); dGrid(1,0); sDash(-1);
```

Set the pen's dash pattern, draw the grid including node numbers, and reset the dash pattern again. The grid is stored in private data structures of PSGRAF.
Drawing the grid is typically a debugging tool to check if the input was correct. In a valid grid, there are no nodes at exactly the same location and grid lines do not fold. While this is not a difficulty with regular grid, typos in the arguments of the semi-regular or of the deformed-regular grid may lead to invalid grids. The situation exacerbates with the completely irregular triangular grids which are typically all hand-made. If you are working in three-dimensions, be aware that the projection of the grid must be a valid grid itself. For instance, a perfectly regular grid, viewed head-on, is not a valid grid for the contouring operator. Since it currently does not check the validity of the grid, the operator will fail unpredictably.

```
deleteGrid();
```

Delete the grid and release the associated private memory. This is only necessary if another grid is to be generated. If the old grid has not been deleted, this will be done automatically by any subsequent grid generator and a corresponding warning message will be displayed.

```
for (i=0;i<6;i++) x[i]=5+pow(2,i)*50/31.;
for (j=0;j<8;j++) y[j]=5+pow(2,j)*50/127.;
gSRGrid(x,6,y,8);
dGrid(0,0);
deleteGrid();
```

After resetting World coordinates and again drawing the axes, create the x- and y-arrays with the respective coordinates of the nodes, generate the corresponding semi-regular grid, and draw it without labeling anything. Notice that the arrays contain the *coordinates*, not the increments between nodes, and thus must be strictly monotonic for the resulting grid to be valid.

```
for (i=0;i<Nx;i++) {
  sphi=sin(i*dphi); cphi=cos(i*dphi);
  for (j=0;j<Ny;j++) {r=0.0001+j*dr; x[i+Nx*j]=r*sphi+5; y[i+Nx*j]=r*cphi+5;}
}
gDRGrid(x,y,Nx,Ny);
```

Generate the coordinates for a deformed regular grid. In contrast to the semi-regular grid, we now have to give the coordinates for each grid point not just for straight grid lines. Hence the length of the arrays `x` and `y` are now the same and equal $\texttt{Nx} \cdot \texttt{Ny}$.

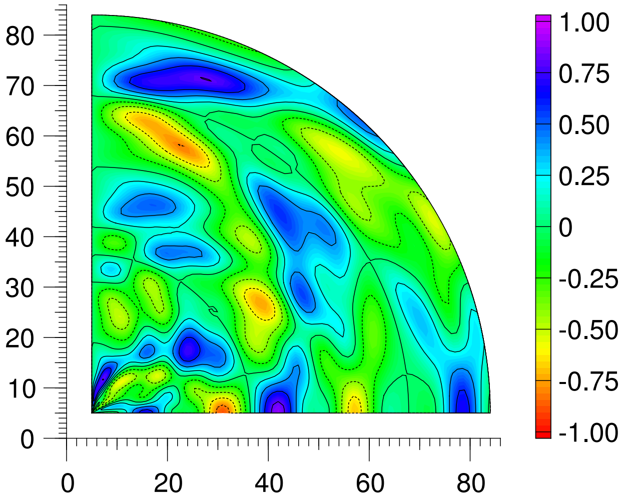
In this example the grid is deformed such that it covers a quarter disk, at least apparently. In reality, this is of course not possible to achieve with a valid grid since the center of the disk is a singularity. We circumvent this by replacing the disk by an annulus with a tiny inner radius, here 0.0001.

```
f=fopen("TGrid.dat","r");
  fscanf(f,"%i %i",&Np,&Nl);
  for (i=0;i<Np;i++) fscanf(f,"%i %lf %lf",&j,x+i,y+i);
  b=(int*)malloc(Nl*sizeof(int));
  e=(int*)malloc(Nl*sizeof(int));
  for (i=0;i<Nl;i++) fscanf(f,"%i %i %i",&j,b+i,e+i);
fclose(f);

gTGrid(x,y,Np,b,e,Nl);
sText("Helvetica",9,'C','C'); dGrid(1,0);
sText("Helvetica",4,'C','C'); dGrid(0,1);
```

Finally, the definition of a completely irregular triangular grid is read from file `TGrid.dat` and the corresponding grid is generated. Since such a file is typically hand-made, we facilitate checking the grid by labeling its nodes as well as its lines.

Notice that the operator `gTGrid` is not particularly efficient – it searches the whole grid for assembling the neighboring information – and that it should only be used for rather small grids, probably with less than a few hundred nodes.

Contours in Two Dimensions



The code `examples/contours.cpp` contours a data set defined on a quarter disk by drawing a few contour lines and filling the area between them with a near-continuous transition of colors. Finally, it draws the corresponding legend.

```
#define Nx       80                          // #nodes in x-direction
#define Ny       80                          // #nodes in x-direction
#define DX       60                          // size of drawing [mm]
#define NCTR     65                          // #contour levels
```

⋮

```
int i,j;
double *x,*y,*z,                             // data set: height z at (x,y)
       ctr[NCTR],                            // values of contour lines
       c0[NCTR+1],c1[NCTR+1],c2[NCTR+1],  // color components of contour fill
       sphi,cphi,dphi=1.5707963/(Nx-1),r,dr=1;
struct dvec orig;
```

Define some constants and variables. Notice that there are `NCTR` contour lines but `NCTR` + 1 contour fills.

```
for (i=0;i<NCTR;i++) {
  ctr[i]=(i+1)*2./(NCTR+1)-1.;
  c0[i]=0.8*(i+1)/(NCTR+1); c1[i]=c2[i]=1;
}
c0[NCTR]=0.8; c1[NCTR]=c2[NCTR]=1;
```

After setting the origin `orig` and allocating memory for the arrays `x`, `y`, and `z`, specify the contour values and the associated fill color. Notice that a contour fill refers to the area that represents values smaller than the corresponding contour value, except for the last fill which refers to the area with larger values. As specified later, the color components refer to the HSB color space and we are using full saturation and brightness. Hue varies only between 0 and 0.8 to prevent the ambiguity that results from the cyclicity of the hue.

```
for (i=0;i<Nx;i++) {
  sphi=sin(i*dphi); cphi=cos(i*dphi);
  for (j=0;j<Ny;j++) {
    r=0.0001+j*dr;
    x[i+Nx*j]=r*sphi+5; y[i+Nx*j]=r*cphi+5;
    z[i+Nx*j]=0.5*sin(0.2*r)*sin(5*i*dphi) +0.3*sin(0.3*r)*sin(7*i*dphi)
             +0.2*sin(0.5*r)*sin(11*i*dphi)+0.1*sin(0.7*r)*sin(13*i*dphi);
  }
}
```

Create the data set: the grid is the same as the deformed regular grid described on page 40 and the “height” `z` is created by adding a few periodic components.

```
sColorSpace("HSB");
sContours(ctr,NCTR,c0,c1,c2);
gDRGrid(x,y,Nx,Ny);
dContours(z,1,0);
```

After requesting a paper, initializing World coordinates and drawing axes, we set the color space and the contour values together with the corresponding fills, generate the deformed regular grid, and finally contour the array `z` by filling the corresponding areas `filled` = 1 but not drawing the contour lines `framed` = 0. Notice that `z` is a one-dimensional array that is arranged identically to `x` and `y`. The correct arrangement is obvious in this example but must also be followed when more regular grids are defined.

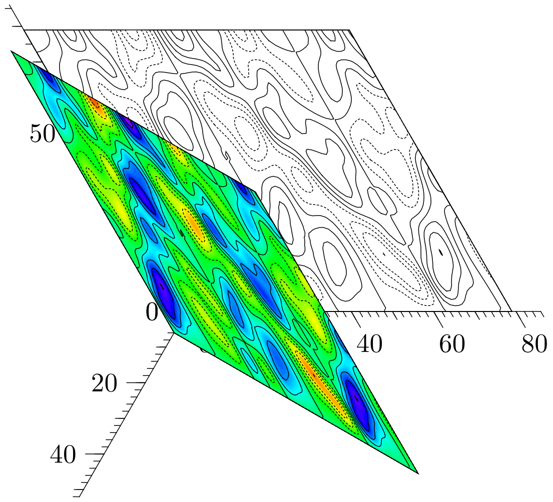
```
dLegend(DX+4,0,-2,DX,0,8);
```

Draw a vertical legend, `horiz` = 0, for the *currently active set of contour values and fills* and label every 8th contour level. The corner of the legend is at $(\mathtt{DX}+4, 0)$, its width is -2 mm and its height `DX` mm. The negative width causes the labeling to move to the right of the color bar and the corner to be at the lower right.

```
sColorSpace("G");
for (i=0;i<5;i++) ctr[i]=i*0.2;
sContours(ctr,5,c0); dContours(z,0,1);
for (i=0;i<4;i++) ctr[i]=-0.8+i*0.2;
sDash(1); sContours(ctr,4,c0); dContours(z,0,1); sDash(-1);
```

Reset the color space to gray, set a few positive contour values, and draw the corresponding contour lines – `filled` = 0, `framed` = 1 – with the current pen (solid black line). Next set a few negative contour values – notice that the must be ordered with increasing values – set the dash pattern and again draw the contour lines. This illustrates that contour lines are always drawn with the current pen.

Contours in Three Dimensions



The code `examples/contours3.cpp` contours a data set defined on an inclined plane and also projects the contour lines into the xy-plane.

```
struct dvec X0,X1,X2;
```

Much the same constants and variables are defined as in the contouring example on page 42 except that a few more vectors are introduced. Their dimension is subsequently set to 3.

```
sXWorldCoord(0,NX+10,0,DX); sYWorldCoord(0,NY+10,0,DX); sZWorldCoord(0,NX+10,0,DX);
X0.x=0;    X0.y=0;   X0.z=0;
X1.x=1;    X1.y=1;   X1.z=1;
X2.x=-1;   X2.y=2;   X2.z=-1;
sView(X0,X1,X2);
```

World coordinates are set in three-dimensional space and the view is definied by the origin `X0`, the direction `X1` from the observer to the origin, and by the direction `X2` that points to the right.

```
dXAxis(X0,86,1);
dYAxis(X0,86,1);
dZAxis(X0,52,1);
```

Draw all three axes starting at the origin.

```
X1.x=1; X1.y=0; X1.z=0;
X2.x=0; X2.y=1; X2.z=0;
gRGrid(X0,X1,X2,NX,NY);
```

Define a regular grid with corner `X0` (still the origin) and grid vectors `X1` and `X2`. Notice that this grid lies in the xy-plane.

```
for (i=0;i<5;i++) ctr[i]=i*0.2;
sContours(ctr,5,c0); dContours(z,0,1);
for (i=0;i<4;i++) ctr[i]=-0.8+i*0.2;
sDash(1); sContours(ctr,4,c0); dContours(z,0,1); sDash(-1);
dGridBoundary(0,0);
deleteGrid();
```

Define and draw contour values in much the same way as for example on page 42, draw the grid boundary without any labeling, and remove the grid.

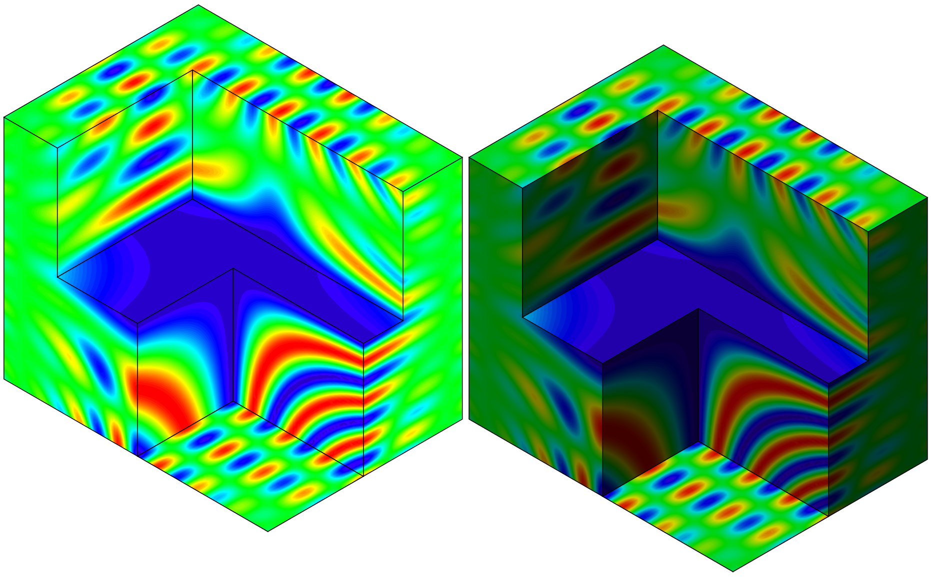
```
X0.x=0; X0.y=0; X0.z=6;
X1.x=1; X1.y=0; X1.z=0;
X2.x=0; X2.y=1; X2.z=0.5;
gRGrid(X0,X1,X2,NX,NY);
dContours(z,1,0);
```

After setting the color space to HSB and defining contour values and fills as in the example above, define a new regular grid and draw the contours. Notice that the previously defined grid was the xy-projection of the current one.

```
sColorSpace("G");
for (i=0;i<5;i++) ctr[i]=i*0.2;
sContours(ctr,5,c0); dContours(z,0,1);
for (i=0;i<4;i++) ctr[i]=-0.8+i*0.2;
sDash(1); sContours(ctr,4,c0); dContours(z,0,1); sDash(-1);
dGridBoundary(0,0);
```

Finally draw some contour lines that correspond to the projections drawn before.

Contouring and Slicing Data Blocks



Code `examples/3d-slices.cpp` generates a three-dimensional data block and contours its outer faces. Then, it cuts two sub-blocks and contours their inner faces. Recall that (i) PSGraf has no built-in concept of three-dimensional space and (ii) drawing is always done in a covering mode, hence earlier parts of a figure are covered by later parts. The order of invoking the `dCBlock` operator and whether it used for drawing outer or inner faces is thus crucial for the three-dimensional illusion of the final figure.

```
X0.dim=X1.dim=X2.dim=3;
f=(double***)malloc(NX*sizeof(double**));
for (i=0;i<NX;i++) {
  f[i]=(double**)malloc(NY*sizeof(double*));
  for (j=0;j<NY;j++) f[i][j]=(double*)malloc(NZ*sizeof(double));
}
min=max=0;
for (i=0;i<NX;i++) {
  bx=(double)(i-NX/2)/NX;
  for (j=0;j<NY;j++) {
    by=(double)(j-NY/2)/NX;
    for (k=0;k<NZ;k++) {
      bz=(double)(k-NZ/2)/NX;
      f[i][j][k]=cos(100*bx*by)*cos(40*by*bz)*cos(10*bz*bx);
      if (min>f[i][j][k]) min=f[i][j][k];
      if (max<f[i][j][k]) max=f[i][j][k];
    }
  }
}
```

After defining some variables and data structures, the dimensions of the vectors are initialized, space for the data block `f` is allocated, and it is filled with some funny function. In passing, minimal and maximal values are also determined for later scaling (even though in this particular example, we know that these values are ±1). Instead of this part, you would put your own function here or read in some data.

```
gPaper("fig/3d-slices");
  sXWorldCoord(0,NX,0,DX); sYWorldCoord(0,NY,0,NY*DX/NX); sZWorldCoord(0,NZ,0,NZ*DX/NX);
  X1.x=-1;   X1.y=-1;   X1.z=-1;
  X2.x= 1;   X2.y= 0;   X2.z=-1;
  sView(X1,X2);
```

```
sColorSpace("HSB");
for (i=0;i<NCTR;i++) {
  ctr[i]=min+(i+1)*(max-min)/(NCTR+1);
  c0[i]=MAXHUE*i/(NCTR-1); c1[i]=1; c2[i]=1;
}
c0[NCTR]=MAXHUE; c1[NCTR]=1; c2[NCTR]=0.8;         // "low-light" outlyers
sContours(ctr,NCTR,c0,c1,c2);
illum.dim=0;
```

Next, the drawing paper is initialized and World coordinates are set such that they have the same scale, independent of the size of the block. The view is then set such that we are looking along the space diagonal of a cube, `X1` $= (-1, -1, -1)$, with the y-axis pointing vertically up, `X2` $= (1, 0, -1)$.
The color space is set to HSB and equidistant contour values and fills are calculated using the previously obtained extremal values of the data block. `MAXHUE` has been set to 0.7 in the constants section to prevent cyclical colors. Notice that regions with very large values are particularly marked by reducing the brightness of the corresponding fill. Obviously, any part of the data set can be highlighted in this and similar ways.
Finally, we set the dimension of the illumination vector `illum` to 0 such that a diffuse light source is used for the first object. Notice that a dimension different from 3 suppresses illumination and `illum` need not be assigned any values for its coordinates.

```
sCBlock(0,1,NX,0,NX-1,
        0,1,NY,0,NY-1,
        0,1,NZ,0,NZ-1,1,illum);
dCBlock(f,1,0);
```

The block to be contoured and the desired clipping is then defined by `sCBlock`. The first line of arguments, `0,1,NX,0,NX-1`, refers to the x-direction and first indicates that the origin, in World coordinates, is at 0, the distance between node is 1, and that there are `NX` nodes for the entire block. The next two arguments indicate beginning and end, as node numbers, of the sub-block that is to be contoured. Here, with `0,NX-1`, the x-extent of the sub-block equals that of the entire block. In the following two lines, these five parameters are also given for the y- and for z-direction. Finally, the last two arguments of `sCBlock` specify that the outer faces of the sub-block have to be contoured and give the illumination vector.
The sub-block is contoured by `dCBlock`. The parameters `1,0` indicate that the contours are to be filled but not framed.

```
sCBlock(0,1,NX,(int)(0.2*NX),NX-1,
        0,1,NY,(int)(0.5*NY),NY-1,
        0,1,NZ,(int)(0.3*NZ),NZ-1,0,illum);
dCBlock(f,1,0);

sCBlock(0,1,NX,(int)(0.5*NX),NX-1,
        0,1,NY,0,(int)(0.5*NY),
        0,1,NZ,(int)(0.5*NZ),NZ-1,0,illum);
dCBlock(f,1,0);
```

In the next steps, we cut out two corner blocks by specifying the desired sub-blocks and then contouring their *inner* faces. This completes the first object.

```
sXWorldCoord(0,NX,DX,2*DX); sYWorldCoord(0,NY,0,NY*DX/NX);
sZWorldCoord(0,NZ,-NZ*DX/NX,0);

illum.dim=3; illum.x=-0.3; illum.y=-1; illum.z=-0.6;
```

To draw the second object, we keep the view but change World coordinates, and set the illumination vector `illum` such that the directed light source is above the object slightly oblique such that all face get some light. The block contouring is then repeated identically to the above.

Part III

Appendix

A

The LaTeX style file `PSgraf.sty`

The information to render the graphics created by PSGRAF is contained in the file **`fn.eps`**, where **`fn`** is the name assigned to the drawing paper by **`gPaper`**. If TeX fonts have been selected, the information needed by TeX to typeset them is in the file **`fn.tex`**.
To incorporate the graphics created by PSGRAF into a TeX file, the macro **`\epsffile`** (written by Tomas Rokicki of Radical Eye Software and provided in **`epsf.sty`**) or a similar tool may be employed. If TeX fonts have been used, the style file **`PSgraf.sty`**, which uses the macro **`\epsffile`**, provides the link between LaTeX and the graphics. The central macro is **`\dTeXText`** which guides the TeX-engine to typeset the lettering of the PostScript graphics.

```
%
% PSgraf.sty: LaTeX style file for incorporating PSgraf files into TeX-documents
%
%  Requires epsf.sty (by Tomas Rokicki of Radical Eye Software) or a similar tool for
%  incorporating pure PostScript into TeX.
%
% HISTORY:
%  v3.0.0                                                                       031227kr
%  created                                                                      950307kr
%
%----------------------------------------------------------------------------------------
%
% PSfigure: draw a centered figure above the figure text
%  1: figure reference
%  2: name (extensions .eps and .tex are appended automatically, do not supply them)
%  3: text for figure caption
%  4: text for list of figures
%  5: position on page (hptb)
%  6: input file(s) T : PostScript graphics & TeX text (fn.eps & fn.tex) [default]
%                   E : Encapsulated PostScript        (fn.eps)
%  7: single (s) or double (d) column
%
\input epsf
%
\newcount\PSfl@g
\newdimen\PS@llx\PS@llx= 10000bp    \newdimen\PS@lly\PS@lly= 10000bp
\newdimen\PS@urx\PS@urx=-10000bp    \newdimen\PS@ury\PS@ury=-10000bp
\newdimen\PS@x    \newdimen\xoff    \newdimen\PS@y    \newdimen\yoff
\newdimen\PS@w    \newdimen\PS@h    \newdimen\PS@Ax   \newdimen\PS@Ay
\newbox\PS@box
%
% dlTeXText    1,2 : position (x,y)
%               3 : text
%               4 : rotation angle
%         5,6,7,8 : cx, cy, cos, sin
%               9 : layer
\def\dTeXText#1#2#3#4#5#6#7#8{\dlTeXText{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{2}}
\def\dlTeXText#1#2#3#4#5#6#7#8#9{%
  \PS@x=#1bp\PS@y=#2bp%
  \setbox\PS@box=\hbox{#3}\PS@w=#5\wd\PS@box\PS@h=#6\ht\PS@box%
  \PS@Ax=-#7\PS@w\advance\PS@Ax by#8\PS@h%
  \PS@Ay=-#8\PS@w\advance\PS@Ay by-#7\PS@h%
  \PS@w=\wd\PS@box\PS@h=\ht\PS@box%
  \ifnum\PSfl@g=0%                              calculate bounding box
    \advance\PS@x by\xoff\advance\PS@x by\PS@Ax%
    \advance\PS@y by\yoff\advance\PS@y by\PS@Ay%
    \ifnum#4<91%
      \advance\PS@x by-#8\PS@h\ifdim\PS@llx>\PS@x\PS@llx=\PS@x\fi\advance\PS@x by#8\PS@h%
      \advance\PS@x by #7\PS@w\ifdim\PS@urx<\PS@x\PS@urx=\PS@x\fi%
      \ifdim\PS@lly>\PS@y\PS@lly=\PS@y\fi%
      \advance\PS@y by#8\PS@w\advance\PS@y by#7\PS@h\ifdim\PS@ury<\PS@y\PS@ury=\PS@y\fi%
    \else%
      \ifnum#4<181%
        \ifdim\PS@urx<\PS@x\PS@urx=\PS@x\fi%
        \advance\PS@x by#7\PS@w
        \advance\PS@x by-#8\PS@h\ifdim\PS@llx>\PS@x\PS@llx=\PS@x\fi%
        \advance\PS@y by#7\PS@h\ifdim\PS@lly>\PS@y\PS@lly=\PS@y\fi%
        \advance\PS@y by-#7\PS@h
        \advance\PS@y by#8\PS@w\ifdim\PS@ury<\PS@y\PS@ury=\PS@y\fi%
      \else%
        \ifnum#4<271%
          \advance\PS@x by-#8\PS@h\ifdim\PS@urx<\PS@x\PS@urx=\PS@x\fi%
          \advance\PS@x by#8\PS@h
          \advance\PS@x by#7\PS@w\ifdim\PS@llx>\PS@x\PS@llx=\PS@x\fi%
          \ifdim\PS@ury<\PS@y\PS@ury=\PS@y\fi%
          \advance\PS@y by#8\PS@w
          \advance\PS@y by#7\PS@h\ifdim\PS@lly>\PS@y\PS@lly=\PS@y\fi%
        \else%
          \ifdim\PS@llx>\PS@x\PS@llx=\PS@x\fi%
          \advance\PS@x by#7\PS@w
          \advance\PS@x by-#8\PS@h\ifdim\PS@urx<\PS@x\PS@urx=\PS@x\fi%
          \advance\PS@y by#7\PS@h\ifdim\PS@ury<\PS@y\PS@ury=\PS@y\fi%
          \advance\PS@y by-#7\PS@h
          \advance\PS@y by#8\PS@w\ifdim\PS@lly>\PS@y\PS@lly=\PS@y\fi%
        \fi%
      \fi%
    \fi%
  \else%                                         draw text of layer \PSfl@g
    \ifnum\PSfl@g=#9%
      \advance\PS@x by\xoff\advance\PS@x by\PS@Ax%
      \advance\PS@y by\yoff\advance\PS@y by\PS@Ay%
      \kern\PS@x\raise\PS@y\hbox to0mm{%
       {\setbox0=\hbox to0pt{#3\hss}%
          \special{ps:gsave}%
          \special{ps:currentpoint currentpoint translate #4 neg
                      rotate neg exch neg exch translate}%
          \box0%
          \special{ps:grestore}%
        }%
      }\kern-\PS@x%
    \fi%
  \fi}
\long\def\PSfigure#1#2#3#4#5#6#7{%
  \if#7s\begin{figure}[#5]\else\begin{figure*}[#5]\fi
  \if#6E%
    \hbox to\textwidth{\hfill\epsffile{#2.eps}\hfill}%
  \else%
    \PSfl@g=0\input{#2.tex}%
    \advance\PS@urx by-\PS@llx\advance\PS@ury by-\PS@lly%
    \hfill%
    \raise-\PS@lly\vbox to\PS@ury{\vfill%
      \hbox to\PS@urx{%
        \kern-\PS@llx\hbox to0pt{\PSfl@g=1\input{#2.tex}\hss}%
        \hbox to0pt{\epsffile{#2.eps}\hss}%
        \PSfl@g=2\input{#2.tex}\hss%
      }%
    }%
  \fi%
  \hfill\hskip0pt%
  \if#3\empty\else\caption[#4]{\label{f#1}#3}\fi%
  \if#7s\end{figure}\else\end{figure*}\fi%
}
```

The usage of this style file is illustrated by the following excerpts from the source of this manual. First, the PSgraf.sty file is included, together with some additional definitions contained in man.sty, by specifying it as optional argument to `\documentstyle`.

```
\documentstyle[man,PSgraf]{report}
```

Each of the figures is then included by invoking `\PSfigure` at the appropriate place, e.g., for Figure 2.1 on page 8 the source is

```
%
\PSfigure{f1}{coordinates}
{{\sf PS\small GRAF} knows two different coordinate systems: (i) Paper coordinates, which
are used to navigate on the drawing paper in much the same way as you would do it using a
ruler, and (ii) World coordinates, which facilitate the drawing of data in their natural
coordinate system.}
{}
{tb}
{T}
{s}
%
```

In the text, the figure is referenced as `Figure \ref{f1}` and the files created by PSGRAF are `coordinates.eps` and `coordinates.tex`.

B
The Header Files

The functions and data structures of PSGRAF are divided into a public and a private part. The corresponding definitions are contained in the header files `PSgraf3.h` (public) and `PSgraf_p.h` (private). A typical program includes only `PSgraf3.h`, i.e., all the internal structures are hidden. This adds considerable security to programming as the internal symbolic names used by PSGRAF are protected.
The header `PSgraf_p.h` must be included, however, if PSGRAF is not used as a library and some parts are incorporated into the source code. Even in this case, direct operation on the internal structures that go beyond reading are strongly discouraged, since they may compromise the consistency of data which are used by other PSGRAF functions.

B.1 `PSgraf3.h`

```
/*---------------------------------------------------------------------------------------
VERSION 3.0.1, COPYRIGHT 1991,1992,1995,1997,2000,2003 K. ROTH

PSgraf IS LICENSED FREE OF CHARGE. THEREFORE THIS FILE, AND ALL THE ACCOMPANYING FILES
WHICH IN THEIR ENTIRETY CONSTITUTE PSgraf, ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, WHETHER EXPRESSED OR IMPLIED. YOU ARE RESPONSIBLE FOR ASCERTAINING THE FITNESS OF
PSgraf FOR ANY SPECIFIC USE, AND CONSEQUENTLY YOU ASSUME ALL THE RESPONSIBILITIES AND
COST THAT MAY ARISE FROM USING IT.
---------------------------------------------------------------------------------------*/

/*
PSgraf3.h: header file for PSgraf

history:
  debug dSymbol, added pPW (3.0.1)         040623kr
  simple 3d objects added; introduce struct dvec; C++ (3.0.0)                   031115kr
  encapsulated bitmap preview (2.4.1)                                           011226kr
  arbitrary triangular grids (2.4.0a)                                           001216kr
  added bitmaps and TIFF                                                        971121kr
  delete magnification and sPaperCoord                                          950330kr
  make all coordinates positive (compatibility with other OS)                   950330kr
  variable argument lists for gray/color                                        920819kr
  color added (2.0)                                                             920523kr
  created based on SciGraf for the Macintosh (1.0)                              910608kr

modifications:
*/

/*-------------------------------------------------------------- begin include watcher */
#ifndef PSgrafVersion


/*------------------------------------------------------------------------ definitions */
#define PSgrafVersion 301
#define NONE -1

struct dvec {                   /* vector in space */
  int dim;                              /* dimension */
  double x,y,z;                         /* coordinates */
};

struct color {                  /* color */
  int CS;                               /* color space (G=0, RGB=2, HSB=4, CMYK=8) */
  double c0,c1,c2,c3;                   /* coordinates */
};

/*-------------------------------------------------- vector operations (vectorops.cpp) */
struct dvec identity(struct dvec x);
struct dvec negvec(struct dvec x);
double      norm(struct dvec x);
void        normalize(struct dvec*x);
struct dvec normalize(struct dvec x);
struct dvec addvec(struct dvec a,struct dvec b);
struct dvec subvec(struct dvec a,struct dvec b);
struct dvec scalmult(double a,struct dvec b);
double      dotprod(struct dvec a,struct dvec b);
void        vecprod(struct dvec a,struct dvec b,struct dvec*c);
struct dvec vecprod(struct dvec a,struct dvec b);

/*-------------------------------------------------------- projectors (projectors.cpp) */
struct dvec pPI(struct dvec Xp);
void        pPI(struct dvec Xp,struct dvec*Xi);
void        pPI(double xp,double yp,double zp,double*xi,double*yi);
struct dvec pPW(struct dvec Xp);
void        pPW(struct dvec Xp,struct dvec *Xw);
void        pPW(double xp,double yp,double*xw,double*yw);
void        pPW(double xp,double yp,double zp,double*xw,double*yw,double*zw);
struct dvec pWI(struct dvec Xw);
void        pWI(struct dvec Xw,struct dvec*Xi);
void        pWI(double xw,double yw,double zw,double*xi,double*yi);
struct dvec pWP(struct dvec Xw);
void        pWP(struct dvec Xw,struct dvec*Xp);
void        pWP(double xw,double yw,double*xp,double*yp);
void        pWP(double xw,double yw,double zw,double*xp,double*yp,double*zp);

/*------------------------------------------------------------------------- functions */
void dArrow(char CS,int type,struct dvec X1);
void dArrow(char CS,int type,struct dvec X0,struct dvec X1);
void dArrow(char CS,int type,struct dvec X0,struct dvec X1,struct dvec O);
void dAArrow(char CS,int type,struct dvec X1,const char*text);
void dAArrow(char CS,int type,struct dvec X0,struct dvec X1,const char*text);
void dAArrow(char CS,int type,struct dvec X0,struct dvec X1,struct dvec O,const char*text);
void dBitMap(double x0,double y0,double dpx,double dpy,int Nx,int Ny,
     double*gry);
void dBitMap(double x0,double y0,double dpx,double dpy,int Nx,int Ny,
     double*c0,double*c1,double*c2);
void dBitMap(double x0,double y0,double dpx,double dpy,int Nx,int Ny,
     double*c0,double*c1,double*c2,double*c3);
void dCBlock(double***f,int filled,int framed);
void dContours(double*Z,int filled,int framed);
void dDataPoint(char CS,struct dvec X,struct dvec dX);
void dDataPoint(char CS,double x,double dx,double y,double dy);
void dDensity(char CS,double x0,double y0,double dx,double dy,int Nx,int Ny,
      double*zd,double misval);
void deleteGrid(void);
void dGrid(int Node_numbers,int Line_numbers);
void dGridBoundary(int Node_numbers,int Line_numbers);
void dLegend(double x0,double y0,double dx,double dy,int horiz,int nth);
void dLine(char CS,struct dvec x0,struct dvec x1);
void dLine(char CS,double x0,double y0,double x1,double y1);
void dLine(char CS,double x0,double y0,double z0,double x1,double y1,double z1);
void dMissing(int*missing);
void dNumber(double thenumber);
void dParallel(char CS,struct dvec x0,struct dvec s0,struct dvec s1);
void dParallel(char CS,struct dvec x0,struct dvec s0,struct dvec s1,struct dvec s2);
void dPolygon(char CS,struct dvec*x,int N);
void dPolygon(char CS,double*x,double*y,int N);
void dPolygon(char CS,double*x,double*y,double*z,int N);
void dSymbol(char CS,struct dvec x,double R,int SY);
void dSymbol(char CS,double x,double y,double R,int SY);
void dSymbol(char CS,double x,double y,double z,double R,int SY);
void dText(const char*text);
void dTIFF(double x0,double y0,double*dpx,double*dpy,const char*fn);
void dXAxis(struct dvec X0,double x1,int below);
void dXAxis(double x0,double y0,double x1,int below);
void dXAxis(double x0,double y0,double z0,double x1,int below);
void dYAxis(struct dvec X0,double y1,int atleft);
void dYAxis(double x0,double y0,double y1,int atleft);
void dYAxis(double x0,double y0,double z0,double y1,int atleft);
void dZAxis(struct dvec X0,double z1,int below);
void dZAxis(double x0,double y0,double z1,int below);
void dZAxis(double x0,double y0,double z0,double z1,int below);
void endPS(void);
int gColorSpace();
void gContours(double*ctr,int*N,struct color*c);
void gDRGrid(double*x,double*y,int Nx,int Ny);
void gDRGrid(double*x,double*y,double*z,int Nx,int Ny);
struct color gFill();
void gRGrid(struct dvec x0,struct dvec x1,struct dvec x2,int N1,int N2);
void gRGrid(double x0,double dx,int Nx,double y0,double dy,int Ny);
void gSRGrid(struct dvec x0,struct dvec x1,int N1,struct dvec x2,int N2);
void gSRGrid(double*x,int Nx,double*y,int Ny);
struct color gStroke();
void gTGrid(double*x,double*y,int NN,int*b,int*e,int NL);
void gPaper(char*name);
void gPreview(void);
int  gTCover();
int  interpolate(double Z[],int missing[]);
void iTCover(int NT,int**nodes,double*cgx,double*cgy);
void MinMax(double*X,int N,double*xmin,double*xmax);
void movea(char CS,struct dvec x);
void movea(char CS,double x,double y);
void movea(char CS,double x,double y,double z);
void mover(char CS,struct dvec x);
void mover(char CS,double x,double y);
void mover(char CS,double x,double y,double z);
void PSerror(const char*where,const char*what);
void PSwarning(const char*where,const char*what);
void sArrow(double lhead,double wtail,double whead,char root);
void sCBlock(double x0,double dx,int Nx,int il,int iu,
     double y0,double dy,int Ny,int jl,int ju,
     double z0,double dz,int Nz,int kl,int ku,int outer,struct dvec illum);
void sClipping(char CS,double xmin,double xmax,double ymin,double ymax);
void sClipping();
int  sColorSpace(int cspace);
int  sColorSpace(const char*cspace);
void sContours(double*ctr,int N,struct color*c);
void sContours(double*ctr,int N,double*c0);
void sContours(double*ctr,int N,double*c0,double*c1,double*c2);
void sContours(double*ctr,int N,double*c0,double*c1,double*c2,double*c3);
void sDash(int ip);
void sDash(const char*dpat);
void sDataPoint(int symbol,double rsymbol,double barlength);
void sDensity(double*val,int N,struct color*c);
void sDensity(double*val,int N,double*c0);
void sDensity(double*val,int N,double*c0,double*c1,double*c2);
void sDensity(double*val,int N,double*c0,double*c1,double*c2,double*c3);
void sFill();
void sFill(struct color c);
void sFill(double c0);
void sFill(double c0,double c1,double c2);
void sFill(double c0,double c1,double c2,double c3);
void sIPlane(struct dvec x1,struct dvec x2);
void sIPlane(struct dvec x0,struct dvec x1,struct dvec x2);
void sMinPixel(double pixelsize);
void sNumber(char format,int prec);
void sStroke(struct color c);
void sStroke(double c0);
void sStroke(double c0,double c1,double c2);
void sStroke(double c0,double c1,double c2,double c3);
void sTeXStyle(const char*texsty,int layered);
void sText(const char*font,double size,char hadjust,char vadjust);
void sTextRotation(double theta);
void sThickness(double thick);
void sView(struct dvec view,struct dvec right);
void sView(struct dvec x0,struct dvec view,struct dvec right);
void sXIntervals(double interval0,double interval1,double interval2,int precision);
void sXTicks(double tick0,double tick1,double tick2);
void sXWorldCoord(double wleft,double wright,double pleft,double pright);
void sYIntervals(double interval0,double interval1,double interval2,int precision);
void sYTicks(double tick0,double tick1,double tick2);
void sYWorldCoord(double wlow,double whigh,double plow,double phigh);
void sZIntervals(double interval0,double interval1,double interval2,int precision);
void sZTicks(double tick0,double tick1,double tick2);
void sZWorldCoord(double wlow,double whigh,double plow,double phigh);
void sZWorldCoord();

/*------------------------------------------------------------------ end include watcher */
#endif
```

B.2 PSgraf_p.h

```
/*-----------------------------------------------------------------------------------------
PSgraf_p.h: private part of header file for PSgraf.

history:
  simple 3d objects added; introduce struct dvec; C++ (3.0.0)                    031115kr
  encapsulated bitmap preview (2.4.1)                                            011226kr
  arbitrary triangular grids (2.4.0a)                                            001216kr
  added bitmaps and TIFF                                                         971121kr
  delete magnification and sPaperCoord                                           950330kr
  make all coordinates positive (compatibility with other OS)                    950330kr
  variable argument lists for gray/color                                         920819kr
  color added (2.0)                                                              920523kr
  created based on SciGraf for the Macintosh (1.0)                               910608kr
-----------------------------------------------------------------------------------------*/

#ifdef COM_SYM_DEF
#define EXTERN
#else
#define EXTERN extern
#endif
/*--------------------------------------------------------------------------- definitions */
#include <ctype.h>
#include <string.h>
#include <stdarg.h>
#include <limits.h>
#include <float.h>
#include <math.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include "PSgraf3.h"
#include "tiffio.h"
#include "tiff.h"

#define PSgraflibVersion 301

#define PT_to_MM     2.83465      /* scaling [point] --> [mm]                          */
#define X_stretch    0.56         /* used to estimate the extent of a string...        */
#define Y_stretch    0.73         /* ...in x- and y-direction from curFSize            */
#define CHUNK     4096            /* elements allocated per request for more memory    */
#define Deg_to_Rad   0.017453293  /* pi/180                                            */
#define PS_PIX_SIZE  0.25         /* pixel size (used in dLegend for density plots)    */

/*-------------------------------------------------------------------------- variables */
/*----- filenames, filestatus, number of drawings -------------------------------------*/
EXTERN int PSinitialized;         /* flag for initialization of PSgraf                 */
EXTERN char *paper_name;          /* name of current paper                             */
EXTERN long BoundingBox;          /* position of bounding box in PostScript file       */
EXTERN long PaperShift;           /* position of shifting vector for positive coord.   */
EXTERN long TeXoff;               /* position of xoff & yoff in TeX file               */
EXTERN int PSdrwN;                /* # drawings                                        */
EXTERN int EPSencaps;             /* 0: no preview; 1: add preview (using system call) */

/*----- coordinate systems, current position, bounding box ----------------------------*/
EXTERN double Pa,Pb,Pc,Pd;        /* scaling-rotation matrix for paper coordinates     */
EXTERN double Px,Py;              /* translation vector for paper coordinates          */
EXTERN double Wlx,Wly,Wlz,        /* left end of window in world coordinates           */
              Plx,Ply,Plz,        /* left end of window in paper coordinates           */
              PWx,PWy,PWz;        /* scaling factors [Pr-Pl]/[Wr-Wl]                   */
EXTERN double cpx,cpy;            /* current position (Paper coordinates [mm])         */
EXTERN double Bxll,Byll,Bxur,Byur;/* lower left and upper right corner of bounding box */
EXTERN int z_set;   /* third dimension active        */

/*----- R^3 --> R^2 projection --------------------------------------------------------*/
EXTERN double R32x0,R32y0,R32z0,  /* origin of image plane in normal space             */
              R32x1,R32y1,R32z1,  /* x vector of image plane in normal space           */
              R32x2,R32y2,R32z2;  /* y vector of image plane in normal space           */
EXTERN struct dvec R32_view;      /* unit vector pointing from observer to origin      */

/*----- pen & color -------------------------------------------------------------------*/
EXTERN double Pthick;             /* pen thickness [Paper]                             */
EXTERN int ColorSpace;            /* G=0, RGB=2, HSB=4, CMYK=8                         */
EXTERN double S_c0,F_c0,          /* color component 0 for stroke and fill             */
                                  /*   (F_c0=NONE --> no filling)                      */
              S_c1,F_c1,          /* color component 1 for stroke and fill             */
              S_c2,F_c2,          /* color component 2 for stroke and fill             */
              S_c3,F_c3;          /* color component 3 for stroke and fill             */
                                  /*   (0 if not used)                                 */

/*----- axes --------------------------------------------------------------------------*/
EXTERN double xinterval[3],       /* intervals for i-th order ticks (i=0,..,2)         */
  yinterval[3],zinterval[3];      /*    "                                              */
EXTERN double xTiMaLe[3],         /* length of i-th order ticks                        */
  yTiMaLe[3],zTiMaLe[3];          /*    "                                              */
EXTERN int xprecision,            /* precision for lettering the axes                  */
  yprecision,zprecision;          /*    "                                              */
EXTERN int xauto,yauto,zauto;     /* autom. determination of intervals and precision   */

/*----- clipping ----------------------------------------------------------------------*/
EXTERN int clip_on;               /* clipping on                                       */
EXTERN double clip_xl,clip_xh,    /* clipping bounds in x-direction (PC)               */
  clip_yl,clip_yh;                /* clipping bounds in y-direction (PC)               */

/*----- arrows ------------------------------------------------------------------------*/
EXTERN double arr_lhead,          /* length of head                                    */
  arr_wtail,arr_whead;            /* width of tail and head                            */
EXTERN char arr_root;             /* root of arrow (T,C,H)                             */
EXTERN int arr_tilt;              /* true if annotation is to be tilted                */

/*----- data point --------------------------------------------------------------------*/
EXTERN int dp_sym;                /* symbol for data point                             */
EXTERN double dp_rsym;            /* radius of symbol                                  */
EXTERN double dp_bar;             /* bar length for errors                             */

/*----- fonts & format for numbers ----------------------------------------------------*/
EXTERN int TeXfonts;              /* use TeXfonts: 0: no, 1: yes, 2: no, but file open */
EXTERN char TeXstyle[127];        /* TeX commands to include with every text           */
EXTERN int lay_text,lay_fill;     /* counters for text and fill layers (for TeX only)  */
EXTERN char curFont[63];          /* name of current font                              */
EXTERN double curFSize;           /* current font size [points]                        */
EXTERN char hadj,vadj;            /* horiz.&vert. adjust.: hadj=[L,C,R], vadj=[T,C,B]  */
EXTERN double TextRotation;       /* angle relative to PaperCoordinates                */
EXTERN char nformat;              /* format for writing numbers                        */
EXTERN int precision;             /* precision for writing numbers                     */

/*----- file pointer ------------------------------------------------------------------*/
EXTERN FILE *PSout;               /* PostScript output                                 */
EXTERN FILE *TeXout;              /* TeX output                                        */
```

```
/*----- grid for contouring -----------------------------------------------------------*/
struct line {
  int id;                         /* line number                                       */
  double bx,by,ex,ey;             /* coordinates of beginning (bx,by) and end (ex,ey)  */
  int boundary;                   /* 1: belongs to boundary                            */
  int n[2];                       /* id of beginning (n[0]) and end (n[1]) node        */
  struct line *l[6];              /* pointers to neighboring lines
                                     (only 4 used for triangular grid)                 */
  double xCP,yCP;                 /* coord. of crossing point on this line (if any)    */
  int state;                      /* 0: no CP; 1: unused CP; -1: CP already used       */
};
struct bline {
  struct line *l;                 /* pointer to grid line                              */
  struct bline *path[2];          /* next (path[0]), prev. (path[1]) line on boundary  */
  int state[2];                   /* 0: node[i] not yet used for contourline           */
};
EXTERN int GT_;                   /* grid type 0: topologically regular quadrangular
                                               1: arbitrary triangular                 */
EXTERN struct line *L_;           /* pointer to set of grid lines                      */
EXTERN int NL_;                   /* # lines in grid                                   */
EXTERN struct bline *B_;          /* pointer to beginning of boundary                  */
EXTERN int NB_;                   /* # lines on boundary                               */
EXTERN int N_gn;                  /* # grid nodes                                      */

/*----- contour lines & density plot --------------------------------------------------*/
EXTERN int N_Ctr;                 /* # contours || color transfer points               */
EXTERN double *C_ctr;             /* pointer to contour values                         */
EXTERN struct color *C_col;       /* pointers to colors                                */
EXTERN struct line *first_on_p;   /* first line on polygon                             */
struct polygon {
  double *x,*y;                   /* pointers to arrays with x- and y-coord. of path   */
  int *boundary;                  /* pointer to boundary array (1: belongs to boundary)
                                     *boundary==NULL if this feature is not used       */
  int N;                          /* # points in x and y                               */
  double bxll,byll,bxur,byur;     /* bounding box of polygon (in WC)                   */
  int ictr;                       /* index of polygon's contour level                  */
  int color;                      /* polygon's color (index to C_col, -1 --> white)    */
  struct line *testL;             /* test line to determine if polygon is a hole       */
};
EXTERN struct polygon *P_;        /* set of all contour lines (polygons)               */
EXTERN int NP_;                   /* # contour lines (polygons)                        */
EXTERN double Min_Pix;            /* minimal pixel size (don't draw smaller poly)      */
EXTERN int below_Pix;             /* #polygons below Min_Pix                           */

/*----- element coverage of arbitrary trianangular grid -------------------------------*/
struct Telement {
  int id;                         /* element number                                    */
  int n[3];                       /* node numbers                                      */
  double cg[2];                   /* coordinates of center of gravity                  */
  int l[3];                       /* line numbers                                      */
};
EXTERN struct Telement *T_;       /* set of all triangles                              */
EXTERN int NT_;                   /* # triangular elements                             */

/*----- contoured block ---------------------------------------------------------------*/
EXTERN int CB_Nx,CB_il,CB_iu,     /* #nodes, lower&upper bound of block in x-direction */
  CB_Ny,CB_jl,CB_ju,              /* #nodes, lower&upper bound of block in y-direction */
  CB_Nz,CB_kl,CB_ku;              /* #nodes, lower&upper bound of block in z-direction */
EXTERN double CB_x0,CB_dx,        /* origin and grid constant in x-direction           */
  CB_y0,CB_dy,                    /* origin and grid constant in y-direction           */
  CB_z0,CB_dz;                    /* origin and grid constant in z-direction           */
EXTERN int CB_outer;              /* contour outer skin of block                       */
EXTERN struct dvec CB_illum;     /* direction of illumination (dim=0 -> no shading)   */


/*------------------------------------------------------------------ private functions */
void dP2P(double *x,double *y,int N);
void dP2W(double *x,double *y,int N);
void dP3P(double *x,double *y,double *z,int N);
void dP3W(double *x,double *y,double *z,int N);
void linea(char CS,struct dvec x);
void liner(char CS,struct dvec x);
void liner(char CS,double x,double y);
void l2Pa(double x,double y);
void l2Wa(double x,double y);
void m2Pa(double x,double y);
void m2Wa(double x,double y);
void m3Pa(double x,double y,double z);
void m3Wa(double x,double y,double z);
void updateBB(double x,double y);


#undef COM_SYM_DEF
```

C
Private Implementation Notes

The following implementation notes are intended for those who desire to deal with the intestines of PSGRAF *and should be skipped by all others.*
For those still reading on: these notes are a rather arbitrary and unstructured selection that may be expanded if need arises.

C.1 Layered Text with TeXfonts

The separation of PostScript and TEX prevents the fine interlayering of text and filled regions which partly hide the text. The crude solution implemented with PSGRAF3 is to track text and filled polygons. Text is then separated into a bottom layer, which will be typeset by the TEX-engine before rendering the PostScript and may thus become partially or fully hidden, and into a top layer which will be typeset afterwards. To this end, the two counters `lay_text` and `lay_fill` are introduced. They are initialized by `gPaper` and incremented alternating by `dText`, provided that TeXfonts are used, and by `dP2P`, provided that the polygon is filled. Notice that this suffices since all text and numbers are eventually produced `dText` and all polygons by `dP2P`.
Drawing text with TeXfonts, the text layer is output to file `TeXout` immediately preceeding `\dTeXText`. Upon closing of the drawing by `gPaper` or `endPS`, `TeXout` is reprocessed to incorporated the layer information into the command `\dlTeXText`.
By default, text layering is turned off, `lay_text`$= -1$, and must be turned on by `sTeXstyle`.
For compatibility with drawings produced with earlier versions of PSGRAF, two slightly macros – `\dTeXText` for older text and `\dlTeXText` for the new layered text – have been implemented in the example style file `PSgraf.sty`.

D
Bugs and Fixes

In some very rare cases, `dContours` fails and produces an error message or a single gray level is missing. The reason for this bug is not clear yet. As a temporary remedy, disturb the contour values by a tiny little bit, e.g., by multiplying them with 1.00001. Such a change has typically no graphical consequence.

References

Adobe Systems Incorporated, 1991, *PostScript Language Reference Manual.* Second edition, Addison-Wesley, Reading, Massachusetts.

Kernighan, B.W. and D.M. Ritchie, 1988, *The C Programming Language.* Second edition, Prentice Hall, Englewood Cliffs, New Jersey.

Knuth, D.E., 1988, *The TEXbook.* Addison-Wesley, Reading, Massachusetts.

Lamport, L., 1986, *LATEX: A Document Preparation System.* Addison-Wesley, Reading, Massachusetts.

Radical Eye Software, 1990, *NeXTTEX: An Implementation of TEX for the NeXT Computer.* Box 2081, Stanford, California.

Index