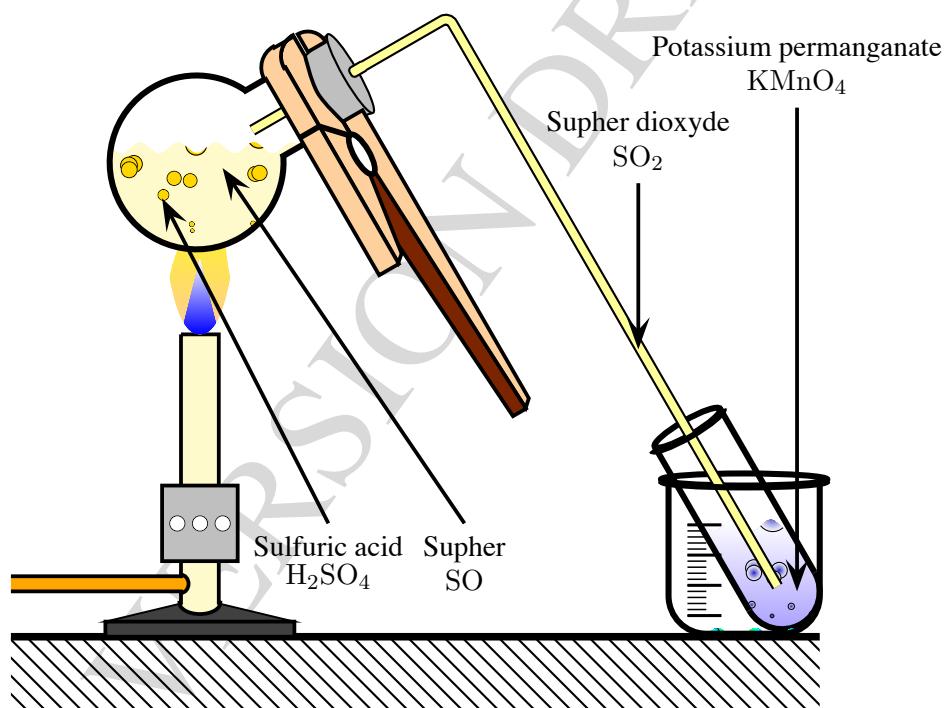


PSTricks
PostScript macros for Generic TeX

User's Guide¹

Timothy Van Zandt

25 July 2003
Version 97



Author's address:
Department of Economics and Political Science
INSEAD
Boulevard de Constance
77305 Fontainebleau Cedex
France

Internet: <tvz@econ.insead.fr>

¹Documentation repackaged in 2003 by Denis Girou <Denis.Girou@idris.fr>

Contents

Welcome to PSTricks	1
Part I The Essentials	3
1 Arguments and delimiters	3
2 Color	4
3 Setting graphics parameters	4
4 Dimensions, coordinates and angles	5
5 Basic graphics parameters	7
Part II Basic graphics objects	9
6 Lines and polygons	9
7 Arcs, circles and ellipses	11
8 Curves	13
9 Dots	15
10 Grids	16
11 Plots	18
Part III More graphics parameters	23
12 Coordinate systems	23
13 Line styles	23
14 Fill styles	25
15 Arrowheads and such	27
16 Custom styles	29
Part IV Custom graphics	31
17 The basics	31
18 Parameters	31
19 Graphics objects	32
20 Safe tricks	35

21	Pretty safe tricks	37
22	For hackers only	38
	Part V Picture Tools	40
23	Pictures	40
24	Placing and rotating whatever	41
25	Repetition	44
26	Axes	46
	Part VI Text Tricks	50
27	Framed boxes	50
28	Clipping	53
29	Rotation and scaling boxes	56
	Part VII Nodes and Node Connections	58
30	Nodes	59
31	Node connections	61
32	Node connections labels: I	70
33	Node connection labels: II	74
34	Attaching labels to nodes	75
35	Mathematical diagrams and graphs	76
36	Complex examples	80
	Part VIII Trees	84
37	Overview	84
38	Tree Nodes	85
39	Tree orientation	88
40	The distance between successors	89
41	Spacing between the root and successors	91
42	Edges	92
43	Edge and node labels	94

44	Framing	98
45	Details	98
46	The scope of parameter changes	101
47	Complex examples	103
	Part IX Filling and Tiling	111
48	Overview	111
49	Usage	112
50	Examples	116
50.1	Kind of tiles	116
50.2	External graphic files	122
50.3	Tiling of characters	123
50.4	“Dynamic” tiling	124
	Part X Three Dimensional Graphics	129
51	Overview	129
52	Usage	129
53	More details	140
	Part XI Other features	145
54	Special coordinates	145
55	Coils and zigzags	148
56	Overlays	150
57	The gradient fill style	151
58	Typesetting text along a path	152
59	Stroking and filling character paths	155
60	Importing EPS files	157
61	Exporting EPS files	159
	Part XII Writing high-level macros	163
62	Overview	163

63 Usage	164
63.1 Integer parameters	165
63.2 Real parameters	166
63.3 String parameters	168
63.4 Boolean parameters	169
63.5 Coordinate arguments	170
63.6 Generic template for a package	184
64 Examples	186
64.1 Basic example : cogged wheels	186
64.2 Advanced example: test tube	188
64.3 Advanced example: Gantt charts	208
64.4 Advanced example: overlapped colored surfaces ¹	214
64.5 Advanced example: three dimensional lighten effect .	225
4 Driver file	230
5 ‘pst-li3d’ L^AT_EX wrapper	230
6 ‘pst-li3d’ code	231
6.1 Preambule	231
6.2 Definition of the parameters	232
6.3 Main macros	234
6.4 Auxiliary macro	236
6.5 Closing	237
7 ‘pst-li3d’ PostScript header file	238
Part XIII Obsolete commands	247
8 Color commands	247
9 Put commands	248
10 Coordinates commands	250
Help	251
A PDF output files	251
A.1 Compilers with internal support of PSTricks	252
A.2 PostScript to PDF converters	252
A.3 PdfT _E X workarounds	252

B	Animated graphics	255
B.1	Pure PStricks graphics	255
B.2	Non pure PStricks graphics	259
C	Generation of code by external programs	261
D	Loops	265
E	Recursion	269
F	Computations	272
G	Storing and retrieving data in arrays	277
H	Random	281
I	Coordinates	287
I.1	Relative coordinates	287
J	Nodes	292
J.1	Getting the coordinates of a node	292
J.2	Positions relative to nodes	293
K	Boxes	294
L	Frequently Asked Questions	297
L.1	How to inactivate the PStricks macros	297
L.2	Monochrome version of all or some graphics	297
L.3	How to improve the rendering of halftones?	298
L.4	Including PostScript code	298
L.5	ConTeXt color macros support	300
L.6	Scope of parameters changes	302
L.7	Defining parameters using macros	302
L.8	Line type connections	304
L.9	Boxes of same sizes for trees and diagrams	306
L.10	Reference points	308
L.11	Dashed and dotted hatched styles	309
L.12	Axes with labels using commas as separators	310
L.13	Alphabetic labels on axes	311
L.14	Two different axes	312
L.15	Polar Plots	314

L.16	Non filled arrows	318
L.17	Lines under a curve	319
L.18	Dashed lines	321

M Troubleshooting

326

Welcome to PStricks

PStricks is a collection of PostScript-based \TeX macros that is compatible with most \TeX macro packages, including PLAIN \TeX , \LaTeX and Con \TeX t, PStricks gives you color, graphics, rotation, trees and overlays. PStricks puts the icing (PostScript) on your cake (\TeX)!

To install PStricks, follow the instructions in the file `read-me.pst` that comes with the PStricks package. Even if PStricks has already been installed for you, give `read-me.pst` a look over.

This *User's Guide* verges on being a reference manual, meaning that it is not designed to be read linearly. Here is a recommended strategy: Finish reading this brief overview of the features in PStricks. Then thumb through the entire *User's Guide* to get your own overview. Return to Part I (Essentials) and read it carefully. Refer to the remaining sections as the need arises.

When you cannot figure out how to do something or when trouble arises, check out the appendices (Help). You just might be lucky enough to find a solution. There is also a \LaTeX file `samples.pst` of samples that is distributed with PStricks. Look to this file for further inspiration.

This documentation is written with \LaTeX . Some examples use \LaTeX specific constructs and some don't. However, there is nothing \LaTeX specific about any of the macros, nor is there anything that does not work with \LaTeX . This package has been tested with PLAIN \TeX , \LaTeX and Con \TeX t and should work with other \TeX macro packages as well.



The main macro file is `pstricks.tex/pstricks.sty`. Each of the PStricks macro files comes with a `.tex` extension and a `.sty` extension; these are equivalent, but the `.sty` extension means that you can include the file name as a \LaTeX package.

There are numerous supplementary macro files. A file, like the one above and the left, is used in this *User's Guide* to remind you that you must input a file before using the macros it contains.

For most PStricks macros, even if you misuse them, you will not get PostScript errors in the output. However, it is recommended that you resolve any \TeX errors before attempting to print your document. A few PStricks macros pass on PostScript errors without warning. Use these with care, especially if you are using a networked printer, because PostScript errors can cause a printer to bomb. Such macros are pointed out in strong terms, using a warning like this one:



Warning: Use macros that do not check for PostScript errors with care. PostScript errors can cause a printer to bomb!

Keep in mind the following typographical conventions in this User's Guide.

- All literal input characters, i.e., those that should appear verbatim in your input file, appear in upright **Helvetica** and **Helvetica-Bold** fonts.
- Meta arguments, for which you are supposed to substitute a value (e.g., *angle*) appear in slanted *Helvetica-Oblique* and **Helvetica-BoldOblique** fonts.
- The main entry for a macro or parameter that states its syntax appears in a large bold font, *except for the optional arguments, which are in medium weight*. This is how you can recognize the optional arguments.
- References to PStricks commands and parameters within paragraphs are set in **Helvetica-Bold**.

The Essentials

1 Arguments and delimiters

Here is some nitty-gritty about arguments and delimiters that is really important to know.

The PStricks macros use the following delimiters:

Curly braces	{arg}
Brackets (only for optional arguments)	[arg]
Parentheses and commas for coordinates	(x,y)
= and , for parameters	par1=val1, ...

Spaces and commas are also used as delimiters within arguments, but in this case the argument is expanded before looking for the delimiters.

Always use a period rather than a comma to denote the decimal point, so that PStricks doesn't mistake the comma for a delimiter.

The easiest mistake to make with the PStricks macros is to mess up the delimiters. This may generate complaints from \TeX or PStricks about bad arguments, or other unilluminating errors such as the following:

- ! Use of \get@coor doesn't match its definition.
- ! Paragraph ended before \pst@addcoor was complete.
- ! Forbidden control sequence found while scanning use of \check@arrow.
- ! File ended while scanning use of \put.

Delimiters are generally the first thing to check when you get errors with a PStricks macro.

Since PStricks macros can have many arguments, it is useful to know that you can leave a space or new line between any arguments, except between arguments enclosed in curly braces. If you need to insert a new line between arguments enclosed in curly braces, put a comment character % at the end of the line.

As a general rule, the first non-space character after a PStricks macro should not be a [or (. Otherwise, PStricks might think that the [or (is actually part of the macro. You can always get around this by inserting a pair {} of braces somewhere between the macro and the [or (.

2 Color

The grayscales

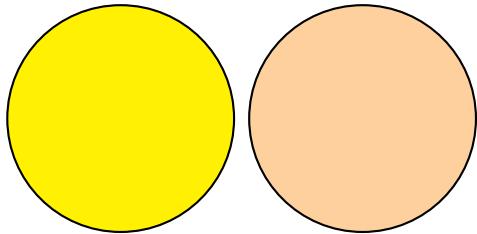
black, darkgray, gray, lightgray, and white,

and the colors

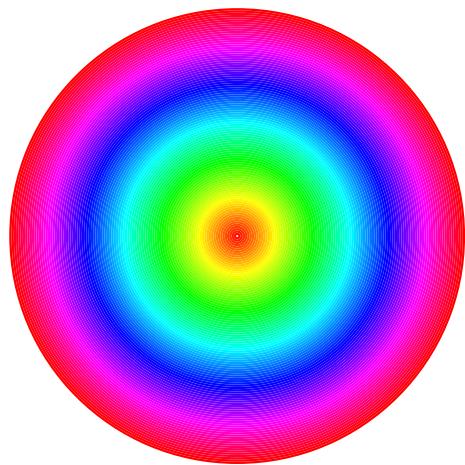
red, green, blue, cyan, magenta, and yellow

are predefined in PStricks.

This means that these names can be used with the graphics objects that are described in later sections.



```
1 \psset{fillstyle=solid}%
2 \pscircle[fillcolor=yellow]{1.5}
3 \definecolor{LightOrange}{cmyk}{0,0.2,0.4,0}%
4 \pscircle[fillcolor=LightOrange]{3.2,0}{1.5}
```



```
1 \psset{unit=3}%
2 \multido{\nHue=0.01+0.01}{100}{%
3   \definecolor{MyColor}{hsb}{\nHue,1,1}%
4   \pscircle[inewidth=0.01, linecolor=MyColor]{\nHue}}
```

3 Setting graphics parameters

PStricks uses a key-value system of graphics parameters to customize the macros that generate graphics (e.g., lines and circles), or graphics combined with text (e.g., framed boxes). You can change the default values of parameters with the command **\psset**, as in

```
1 \psset{fillcolor=yellow}
2 \psset{linecolor=blue, framearc=.3, dash=3pt 6pt}
```

The general syntax is:

```
\psset{par1=value1,par2=value2,... }
```

As illustrated in the examples above, spaces are used as delimiters for some of the values. Additional spaces are allowed only following the comma that separates *par*=*value* pairs (which is thus a good place to start a new line if there are many parameter changes). E.g., the first example is acceptable, but the second is not:

```
1 \psset{fillcolor=yellow, linecolor=blue}  
2 \psset{fillcolor= yellow, linecolor =blue }
```

The parameters are described throughout this *User's Guide*, as they are needed.

Nearly every macro that makes use of graphics parameters allows you to include changes as an optional first argument, enclosed in square brackets. For example,

```
1 \psline[linecolor=green,linestyle=dotted](8,7)
```

draws a dotted, green line. It is roughly equivalent to

```
1 {\psset{linecolor=green,linestyle=dotted}\psline(8,7)}
```

For many parameters, PSTricks processes the value and stores it in a peculiar form, ready for PostScript consumption. For others, PSTricks stores the value in a form that you would expect. In the latter case, this *User's Guide* will mention the name of the command where the value is stored. This is so that you can use the value to set other parameters. E.g.,

```
1 \psset{linecolor=\psfillcolor,doublesep=.5\pslinewidth}
```

However, even for these parameters, PSTricks may do some processing and error-checking, and you should always set them using **\psset** or as optional parameter changes, rather than redefining the command where the value is stored.

4 Dimensions, coordinates and angles

Whenever an argument of a PSTricks macro is a dimension, the unit is optional. The default unit is set by the

unit=*dim*

Default: 1cm

parameter. For example, with the default value of 1cm, the following are equivalent:

```

1 \psset{linewidth=.5cm}
2 \psset{linewidth=.5}

```

By never explicitly giving units, you can scale graphics by changing the value of **unit**.

You can use the default coordinate when setting non-PSTricks dimensions as well, using the commands

\pssetlength{cmd}{dim}
\psaddtolength{cmd}{dim}

where *cmd* is a dimension register (in L^AT_EX parlance, a “length”), and *dim* is a length with optional unit. These are analogous to L^AT_EX’s **\setlength** and **\addtolength**.

Coordinate pairs have the form (x, y) . The origin of the coordinate system is at T_EX’s currentpoint. The command **\SpecialCoor** lets you use polar coordinates, in the form $(r; a)$, where *r* is the radius (a dimension) and *a* is the angle (see below). You can still use Cartesian coordinates. For a complete description of **\SpecialCoor**, see Section 54.

The **unit** parameter actually sets the following three parameters:

xunit=dim	Default: 1cm
yunit=dim	Default: 1cm
runit=dim	Default: 1cm

These are the default units for x-coordinates, y-coordinates, and all other coordinates, respectively. By setting these independently, you can scale the x and y dimensions in Cartesian coordinate unevenly. After changing **yunit** to 1pt, the two **\psline**’s below are equivalent:

```

1 \psset{yunit=1pt}
2 \psline(0cm,20pt)(5cm,80pt)
3 \psline(0,20)(5,80)

```

The values of the **runit**, **xunit** and **yunit** parameters are stored in the dimension registers **\psunit** (also **\psrunit**), **\psxunit** and **\psyunit**.

Angles, in polar coordinates and other arguments, should be a number giving the angle in degrees, by default. You can also change the units used for angles with the command

\degrees[num]

num should be the number of units in a circle. For example, you might use

```

1 \degrees[100]

```

to make a pie chart when you know the shares in percentages. `\degrees` without the argument is the same as

```
1 \degrees[360]
```

The command

radians

is short for

```
\degrees[6.28319]
```

\SpecialCoor lets you specify angles in other ways as well.

5 Basic graphics parameters

The width and color of lines is set by the parameters:

linewidth=dim
linecolor=color

Default: .8pt
Default: black

The **linewidth** is stored in the dimension register `\pslinewidth`, and the **linecolor** is stored in the command `\pslinecolor`.

The regions delimited by open and closed curves can be filled, as determined by the parameters:

fillstyle=style
fillcolor=color

When **fillstyle=none**, the regions are not filled. When **fillstyle=solid**, the regions are filled with **fillcolor**. Other **fillstyle**'s are described in Section 14.

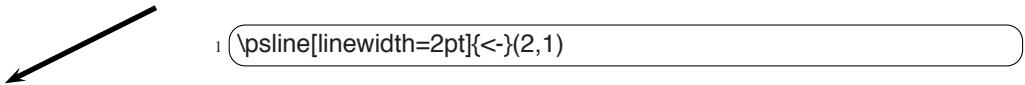
The graphics objects all have a starred version (e.g., `\psframe*`) which draws a solid object whose color is **linecolor**. For example,

```
1 \psellipse*(1,.5)(1,.5)
```

Open curves can have arrows, according to the

arrows=arrows

parameter. If **arrows=-**, you get no arrows. If **arrows=<->**, you get arrows on both ends of the curve. You can also set **arrows=>** and **arrows=<**, if you just want an arrow on the end or beginning of the curve, respectively. With the open curves, you can also specify the arrows as an optional argument enclosed in {} brackets. This should come after the optional parameters argument. E.g.,



Other arrow styles are described in Section 15

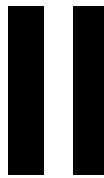
If you set the

showpoints=true/false

Default: false

parameter to **true**, then most of the graphics objects will put dots at the appropriate coordinates or control points of the object.² Section 9 describes how to change the dot style.

²The parameter value is stored in the conditional `\ifshowpoints`.



Basic graphics objects

6 Lines and polygons

The objects in this section also use the following parameters:

linearc=dim

Default: Opt

The radius of arcs drawn at the corners of lines by the **\psline** and **\pspolygon** graphics objects. *dim* should be positive.

framearc=num

Default: 0

In the **\psframe** and the related box framing macros, the radius of rounded corners is set, by default, to one-half *num* times the width or height of the frame, whichever is less. *num* should be between 0 and 1.

cornersize=relative/absolute

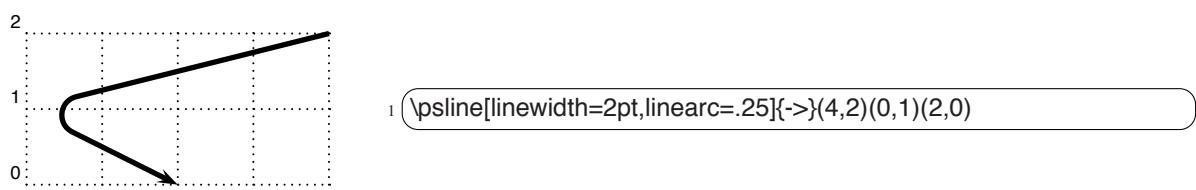
Default: relative

If **cornersize** is relative, then the **framearc** parameter determines the radius of the rounded corners for **\psframe**, as described above (and hence the radius depends on the size of the frame). If **cornersize** is absolute, then the **linearc** parameter determines the radius of the rounded corners for **\psframe** (and hence the radius is of constant size).

Now here are the lines and polygons:

\psline*[par]{arrows}(x0,y0)(x1,y1)…(xn,yn)

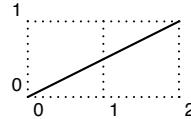
This draws a line through the list of coordinates. For example:



\qline(coor0)(coor1)

This is a streamlined version of **\psline** that does not pay attention to the **arrows** parameter, and that can only draw a single line segment. Note that both coordinates are obligatory, and there is no optional

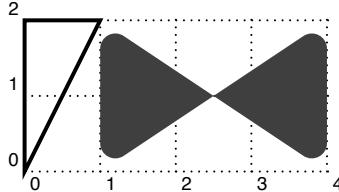
argument for setting parameters (use `\psset` if you need to change the `linewidth`, or whatever). For example:



```
1 \qline(0,0)(2,1)
```

`\pspolygon*[par](x0,y0)(x1,y1)(x2,y2)...(xn,yn)`

This is similar to `\psline`, but it draws a closed path. For example:



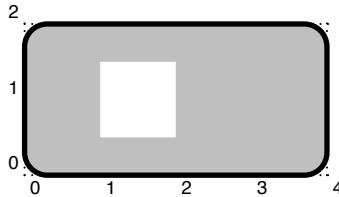
```
1 \pspolygon*[linewidth=1.5pt](0,2)(1,2)
```

```
2 \pspolygon*[lineararc=.2, linecolor=darkgray](1,0)(1,2)(4,0)(4,2)
```

`\psframe*[par](x0,y0)(x1,y1)`

`\psframe` draws a rectangle with opposing corners (x_0, y_0) and (x_1, y_1) .

For example:



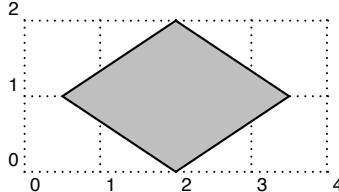
```
1 \psframe[linewidth=2pt, framearc=.3, fillstyle=solid,
```

```
2 fillcolor=lightgray](4,2)
```

```
3 \psframe*[linecolor=white](1,.5)(2,1.5)
```

`\psdiamond*[par](x0,y0)(x1,y1)`

`\psdiamond` draws a diamond centered at (x_0, y_0) , and with the half width and height equal to x_1 and y_1 , respectively.



```
1 \psdiamond*[framearc=.3, fillstyle=solid,
```

```
2 fillcolor=lightgray](2,1)(1.5,1)
```

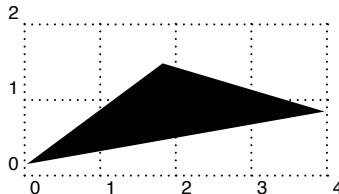
The diamond is rotated about the center by

`gangle=angle`

Default: 0

`\pstriangle*[par](x0,y0)(x1,y1)`

`\pstriangle` draws an isosceles triangle with the base centered at (x_0, y_0) , and with width (base) and height equal to x_1 and y_1 , respectively.

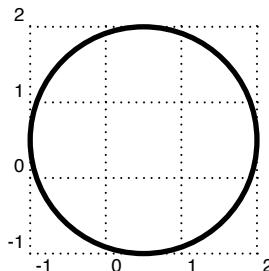


```
1 \pstriangle*[gangle=10](2,.5)(4,1)
```

7 Arcs, circles and ellipses

\pscircle*[par](x0,y0){radius}

This draws a circle whose center is at $(x0, y0)$ and that has radius *radius*. For example:



1 \pscircle*[linewidth=2pt](.5,.5){1.5}

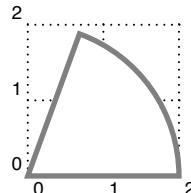
\qdisk(coor){radius}

This is a streamlined version of **\pscircle***. Note that the two arguments are obligatory and there is no parameters arguments. To change the color of the disks, you have to use **\psset**:

1 \psset{linecolor=gray}
2 \qdisk(2,3){4pt}

\pswedge*[par](x0,y0){radius}{angle1}{angle2}

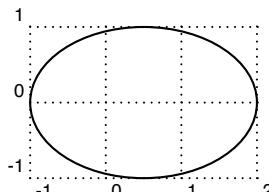
This draws a wedge whose center is at $(x0, y0)$, that has radius *radius*, and that extends counterclockwise from *angle1* to *angle2*. The angles must be specified in degrees. For example:



1 \pswedge*[linecolor=gray,linewidth=2pt,fillstyle=solid]{2}{0}{70}

\psellipse*[par](x0,y0)(x1,y1)

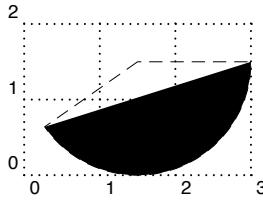
$(x0, y0)$ is the center of the ellipse, and $x1$ and $y1$ are the horizontal and vertical radii, respectively. For example:



1 \psellipse*[fillcolor=lightgray]{(.5,0)(1.5,1)}

\psarc*[par]{arrows}(x,y){radius}{angleA}{angleB}

This draws an arc from *angleA* to *angleB*, going counter clockwise, for a circle of radius *radius* and centered at (x, y) . You must include either the arrows argument or the (x, y) argument. For example:



```
1 \psarc*[showpoints=true](1.5,1.5){1.5}{215}{0}
```

See how **showpoints=true** draws a dashed line from the center to the arc; this is useful when composing pictures.

\psarc also uses the parameters:

arcsepA=dim

Default: Opt

angleA is adjusted so that the arc would just touch a line of width *dim* that extended from the center of the arc in the direction of *angleA*.

arcsepB=dim

Default: Opt

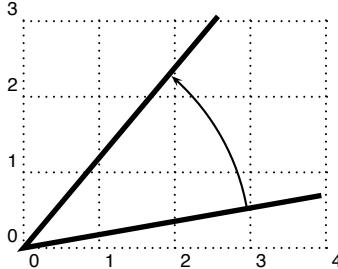
This is like **arcsepA**, but *angleB* is adjusted.

arcsep=dim

Default: 0

This just sets both **arcsepA** and **arcsepB**.

These parameters make it easy to draw two intersecting lines and then use **\psarc** with arrows to indicate the angle between them. For example:



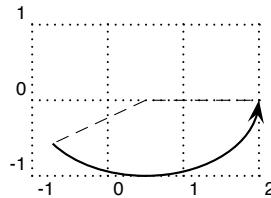
```
1 \SpecialCoor
2 \psline[linewidth=2pt](4;50)(0,0)(4;10)
3 \psarc[arcsepB=2pt]{->}(3){10}{50}
```

\psarcn*[par]{arrows}(x,y){radius}{angleA}{angleB}

This is like **\psarc**, but the arc is drawn *clockwise*. You can achieve the same effect using **\psarc** by switching *angleA* and *angleB* and the arrows.³

\psellipticarc*[par]{arrows}(x0,y0)(x1,y1){angleA}{angleB}

This draws an elliptic from *angleA* to *angleB*, going counter clockwise, with (x_0, y_0) the center of the ellipse and *x1* and *y1* the horizontal and vertical radii, respectively. For example:



```
1 \psellipticarc[showpoints=true,arrowscale=2]{->}(.5,0)(1.5,1){215}{0}
```

³However, with **\pscustom** graphics object, described in Part IV, **\psarcn** is not redundant.

See how **showpoints=true** draws a dashed line from the center to the arc; this is useful when composing pictures.

Like **\psarc**, **\psellipticarc** use the **arcsep/arcsepA/arcsepB** parameters.

Unlike **\psarc**, **\psellipticarc** use the **dimen** parameter.

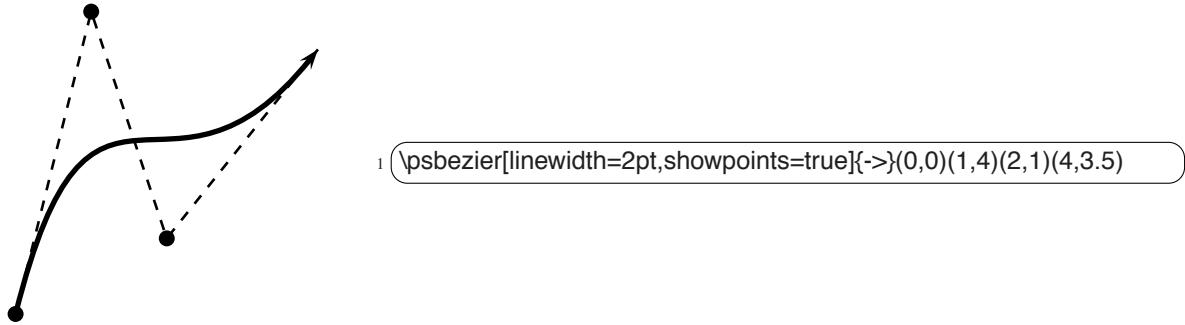
\psellipticarcn*[par]{arrows}(x0,y0)(x1,y1){angleA}{angleB}

This is like **\psellipticarc**, but the arc is drawn *clockwise*. You can achieve the same effect using **\psellipticarc** by switching *angleA* and *angleB* and the arrows.⁴

8 Curves

\psbezier*[par]{arrows}(x0,y0)(x1,y1)(x2,y2)(x3,y3)

\psbezier draws a bezier curve with the four control points. The curve starts at the first coordinate, tangent to the line connecting to the second coordinate. It ends at the last coordinate, tangent to the line connecting to the third coordinate. The second and third coordinates, in addition to determining the tangency of the curve at the endpoints, also “pull” the curve towards themselves. For example:

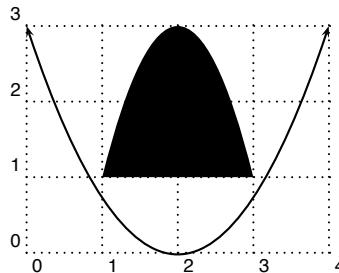


showpoints=true puts dots in all the control points, and connects them by dashed lines, which is useful when adjusting your bezier curve.

\parabola*[par]{arrows}(x0,y0)(x1,y1)

Starting at (x_0, y_0) , **\parabola** draws the parabola that passes through (x_0, y_0) and whose maximum or minimum is (x_1, y_1) . For example:

⁴However, with **\pscustom** graphics object, described in Part IV, **\psellipticarcn** is not redundant.



```

1 \parabola*(1,1)(2,3)
2 \psset{xunit=.01}
3 \parabola{<->}(400,3)(200,0)

```

The next three graphics objects interpolate an open or closed curve through the given points. The curve at each interior point is perpendicular to the line bisecting the angle ABC, where B is the interior point, and A and C are the neighboring points. Scaling the coordinates *does not* cause the curve to scale proportionately.

The curvature is controlled by the following parameter:

curvature=num1 num2 num3

Default: 1 .1 0

You have to just play around with this parameter to get what you want. Individual values outside the range -1 to 1 are either ignored or are for entertainment only. Below is an explanation of what each number does. A, B and C refer to three consecutive points.

Lower values of *num1* make the curve tighter.

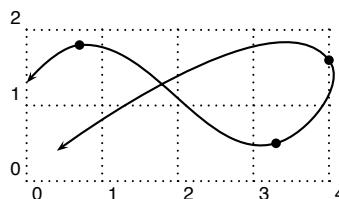
Lower values of *num2* tighten the curve where the angle ABC is greater than 45 degrees, and loosen the curve elsewhere.

num3 determines the slope at each point. If *num3*=0, then the curve is perpendicular at B to the bisection of ABC. If *num3*=-1, then the curve at B is parallel to the line AC. With this value (and only this value), scaling the coordinates causes the curve to scale proportionately. However, positive values can look better with irregularly spaced coordinates. Values less than -1 or greater than 2 are converted to -1 and 2, respectively.

Here are the three curve interpolation macros:

\pscurve*[par]{arrows}(x1,y1)... (xn,yn)

This interpolates an open curve through the points. For example:



```

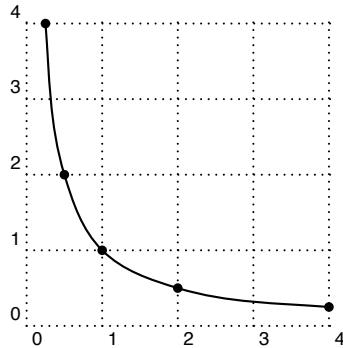
1 \pscurve[showpoints=true]{<->}(0,1.3)(0.7,1.8)
2 (3.3,0.5)(4,1.6)(0.4,0.4)

```

Note the use of **showpoints=true** to see the points. This is helpful when constructing a curve.

\psecurve*[par]{arrows}(x1,y1)... (xn,yn)

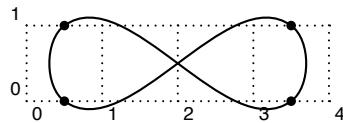
This is like **\pscurve**, but the curve is not extended to the first and last points. This gets around the problem of trying to determine how the curve should join the first and last points. The **e** has something to do with “endpoints”. For example:



```
1 \psecurve[showpoints=true](.125,8)(.25,4)(.5,2)
2 (1,1)(2,.5)(4,.25)(8,.125)
```

\psccurve*[par]{arrows}(x1,y1)... (xn,yn)

This interpolates a closed curve through the points. **c** stands for “closed”. For example:



```
1 \psccurve[showpoints=true]
2 (.5,0)(3.5,1)(3.5,0)(.5,1)
```

9 Dots

The graphics objects

\psdot*[par](x1,y1)
\psdots*[par](x1,y1)(x2,y2)... (xn,yn)

put a dot at each coordinate.

What a “dot” is depends on the value of the

dotstyle=style

Default: *

parameter. This also determines the dots you get when **showpoints=true**.

The dot styles are also pretty intuitive:

<i>Style</i>	<i>Example</i>	<i>Style</i>	<i>Example</i>
*	• • • • •	square	□ □ □ □ □
o	○ ○ ○ ○ ○	Bsquare	▣ ▣ ▣ ▣ ▣
Bo	○ ○ ○ ○ ○	square*	■ ■ ■ ■ ■
x	× × × × ×	diamond	◊ ◊ ◊ ◊ ◊
+	+ + + + +	Bdiamond	◊ ◊ ◊ ◊ ◊
B+	+ + + + +	diamond*	◆ ◆ ◆ ◆ ◆
asterisk	* * * * *	triangle	△ △ △ △ △
Basterisk	* * * * *	Btriangle	△ △ △ △ △
oplus	⊕ ⊕ ⊕ ⊕ ⊕	triangle*	▲ ▲ ▲ ▲ ▲
otimes	⊗ ⊗ ⊗ ⊗ ⊗	pentagon	◎ ◎ ◎ ◎ ◎
		Bpentagon	◎ ◎ ◎ ◎ ◎
Bl		pentagon*	★ ★ ★ ★ ★

Except for diamond, the center of dot styles with a hollow center is colored **fillcolor**.

Here are the parameters for changing the size and orientation of the dots:

dotsize=*dim* ‘*num*’

Default: 2pt 2

The diameter of a circle or disc is *dim* plus *num* times **linewidth** (if the optional *num* is included). The size of the other dots styles is similar (except for the size of the | dot style, which is set by the **tbarsize** parameter described on page 29).

dotscale=*num1* ‘*num2*’

Default: 1

The dots are scaled horizontally by *num1* and vertically by *num2*. If you only include *num1*, the arrows are scaled by *num1* in both directions.

dotangle=*angle*

Default: 0

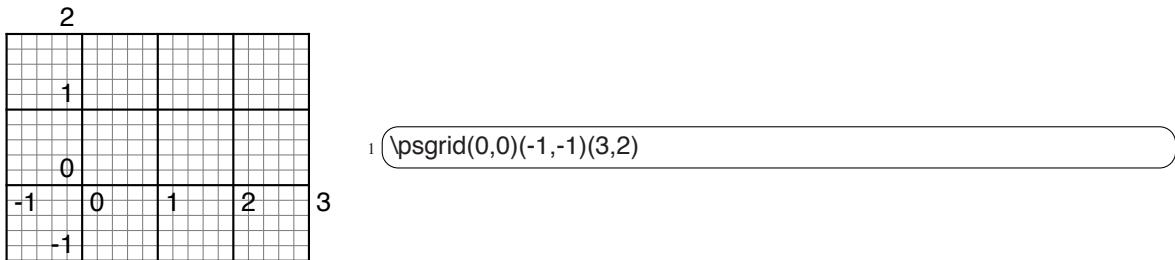
After setting the size and scaling the dots, the dots are rotated by *angle*.

10 Grids

PSTricks has a powerful macro for making grids and graph paper:

\psgrid(x0,y0)(x1,y1)(x2,y2)

\psgrid draws a grid with opposing corners (x_1, y_1) and (x_2, y_2) . The intervals are numbered, with the numbers positioned at x_0 and y_0 . The coordinates are always interpreted as Cartesian coordinates. For example:

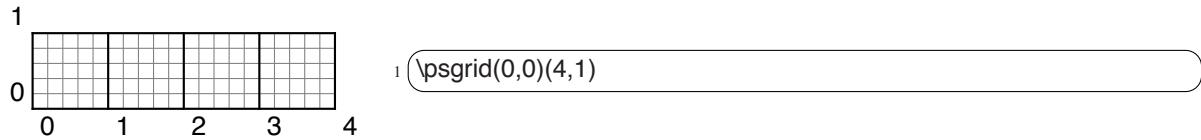


(Note that the coordinates and label positioning work the same as with **\psaxes**.)

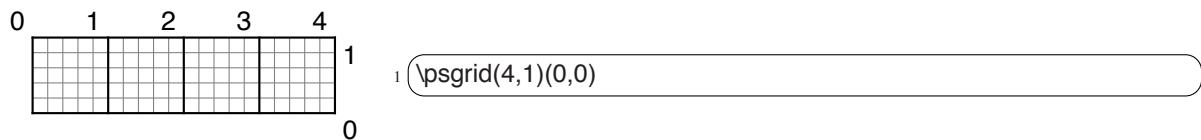
The main grid divisions occur on multiples of **xunit** and **yunit**. Subdivisions are allowed as well. Generally, the coordinates would be given as integers, without units.

If the $(x0, y0)$ coordinate is omitted, $(x1, y1)$ is used. The default for $(x1, y1)$ is $(0,0)$. If you don't give any coordinates at all, then the coordinates of the current **\pspicture** environment are used or a 10×10 grid is drawn. Thus, you can include a **\psgrid** command without coordinates in a **\pspicture** environment to get a grid that will help you position objects in the picture.

The main grid divisions are numbered, with the numbers drawn next to the vertical line at $x0$ (away from $x2$) and next to the horizontal line at $y1$ (away from $y2$). $(x1, y1)$ can be any corner of the grid, as long as $(x2, y2)$ is the opposing corner, you can position the labels on any side you want. For example, compare



and



The following parameters apply only to **\psgrid**:

gridwidth=dim

Default: .8pt

The width of grid lines.

gridcolor=color

Default: black

The color of grid lines.

griddots=num

Default: 0

If *num* is positive, the grid lines are dotted, with *num* dots per division.

gridlabels=dim **Default: 10pt**

The size of the numbers used to mark the grid.

gridlabelcolor=color **Default: black**

The color of the grid numbers.

subgriddiv=int **Default: 5**

The number of grid subdivisions.

subgridwidth=dim **Default: .4pt**

The width of subgrid lines.

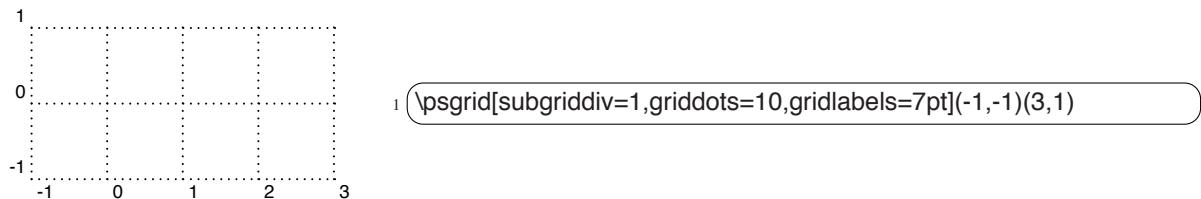
subgridcolor=color **Default: gray**

The color of subgrid lines.

subgriddots=num **Default: 0**

Like **griddots**, but for subdivisions.

Here is a familiar looking grid which illustrates some of the parameters:



Note that the values of **xunit** and **yunit** are important parameters for **\psgrid**, because they determine the spacing of the divisions. E.g., if the value of these is **1pt**, and then you type

```
1 \psgrid(0,0)(10in,10in)
```

you will get a grid with 723 main divisions and 3615 subdivisions! (Actually, **\psgrid** allows at most 500 divisions or subdivisions, to limit the damage done by this kind of mistake.) Probably you want to set **unit** to **.5in** or **1in**, as in

```
1 \psgrid[unit=.5in](0,0)(20,20)
```

11 Plots



The plotting commands described in this part are defined in **pst-plot.tex** / **pst-plot.sty**, which you must load first (for various simple and more elaborated examples, you can refer to the two articles published by Herbert Voß

and Laura E. Jackson in *Die TeXnische Komödie* in 2002 [33] and [81] (in German)).⁵

The **\psdots**, **\psline**, **\pspolygon**, **\pscurve**, **\psecurve** and **\psccurve** graphics objects let you plot data in a variety of ways. However, first you have to generate the data and enter it as coordinate pairs (x, y). The plotting macros in this section give you other ways to get and use the data. (Section 26 tells you how to generate axes.)

To parameter

plotstyle=style

Default: **line**

determines what kind of plot you get. Valid styles are dots, line, polygon, curve, ecurve, ccurve. E.g., if the **plotstyle** is polygon, then the macro becomes a variant of the **\pspolygon** object.

You can use arrows with the plot styles that are open curves, but there is no optional argument for specifying the arrows. You have to use the **arrows** parameter instead.



Warning: No PostScript error checking is provided for the data arguments. Read Appendix L.4 before including PostScript code in the arguments.

There are system-dependent limits on the amount of data \TeX and PostScript can handle. You are much less likely to exceed the PostScript limits when you use the line, polygon or dots plot style, with **showpoints=false**, **linearc=0pt**, and no arrows.

Note that the lists of data generated or used by the plot commands cannot contain units. The values of **\psxunit** and **\psyunit** are used as the unit.

\fileplot*[par]{file}

\fileplot is the simplest of the plotting functions to use. You just need a file that contains a list of coordinates (without units), such as generated by Mathematica or other mathematical packages. The data can be delimited by curly braces {}, parentheses (), commas, and/or white space. Bracketing all the data with square brackets [] will significantly speed up the rate at which the data is read, but there are system-dependent limits on how much data \TeX can read like this in one chunk. (The [must go at the beginning of a line.) The file should not contain anything else (not even **\endinput**), except for comments marked with %.

⁵For plotting commands in three dimensions, see the ‘pst-3dplot’ contribution package [54] from Herbert Voß.

\fileplot only recognizes the line, polygon and dots plot styles, and it ignores the **arrows**, **linearc** and **showpoints** parameters. The **\listplot** command, described below, can also plot data from file, without these restrictions and with faster TeX processing. However, you are less likely to exceed PostScript's memory or operand stack limits with **\fileplot**.

If you find that it takes TeX a long time to process your **\fileplot** command, you may want to use the **\PSTtoEPS** command described on page 160. This will also reduce TeX's memory requirements.

\dataplot*[par]{*commands*}

\dataplot is also for plotting lists of data generated by other programs, but you first have to retrieve the data with one of the following commands:

```
\savedata{command} [data]
\readdata{command} {file}
```

data or the data in *file* should conform to the rules described above for the data in **\fileplot** (with **\savedata**, the data must be delimited by [], and with **\readdata**, bracketing the data with [] speeds things up). You can concatenate and reuse lists, as in

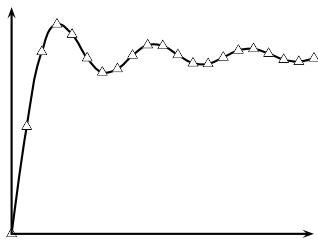
```
1 \readdata{\foo}{foo.data}
2 \readdata{\bar}{bar.data}
3 \dataplot{\foo\bar}
4 \dataplot[origin={0,1}]{\bar}
```

The **\readdata** and **\dataplot** combination is faster than **\fileplot** if you reuse the data. **\fileplot** uses less of TeX's memory than **\readdata** and **\dataplot** if you are also use **\PSTtoEPS**.

Here is a plot of $\text{Integral}(\sin(x))$. The data was generated by Mathematica, with

```
1 Table[{x,N[Sin[Integral[x]]]}, {x,0,20}]
```

and then copied to this document.



```
1 \psset{xunit=.2cm,yunit=1.5cm}
2 \savedata{\mydata}{%
3   {{0, 0}, {1., 0.946083}, {2., 1.60541}, {3., 1.84865}, {4., 1.7582},
4   {5., 1.54993}, {6., 1.42469}, {7., 1.4546}, {8., 1.57419},
5   {9., 1.66504}, {10., 1.65835}, {11., 1.57831}, {12., 1.50497},
6   {13., 1.49936}, {14., 1.55621}, {15., 1.61819}, {16., 1.6313},
7   {17., 1.59014}, {18., 1.53661}, {19., 1.51863}, {20., 1.54824}}]
8 \dataplot[plotstyle=curve,showpoints=true,
9   dotstyle=triangle]{\mydata}
10 \psline{<->}(0,2)(0,0)(20,0)
```

\listplot*[par]{list}

\listplot is yet another way of plotting lists of data. This time, *list* should be a list of data (coordinate pairs), delimited only by white space. *list* is first expanded by TeX and then by PostScript. This means that *list* might be a PostScript program that leaves on the stack a list of data, but you can also include data that has been retrieved with **\readadda** and **\dataplot**. However, when using the line, polygon or dots plotstyles with **showpoints=false**, **linearc=0pt** and no arrows, **\dataplot** is much less likely than **\listplot** to exceed PostScript's memory or stack limits. In the preceding example, these restrictions were not satisfied, and so the example is equivalent to when **\listplot** is used:

```

1 ...
2 \listplot[plotstyle=curve,showpoints=true,
3   dotstyle=triangle]{\mydata}
4 ...

```

\psplot*[par]{x_{min}}{x_{max}}{function}

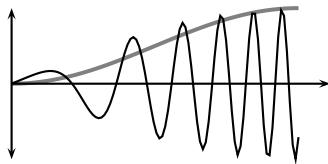
\psplot can be used to plot a function $f(x)$, if you know a little PostScript. *function* should be the PostScript code for calculating $f(x)$. Note that you must use x as the dependent variable. PostScript is not designed for scientific computation, but **\psplot** is good for graphing simple functions right from within TeX. E.g.,

```

1 \psplot[plotpoints=200]{0}{720}{x sin}

```

plots $\sin(x)$ from 0 to 720 degrees, by calculating $\sin(x)$ roughly every 3.6 degrees and then connecting the points with **\psline**. Here are plots of $\sin(x) \cos((x/2)^2)$ and $\sin^2(x)$:



```

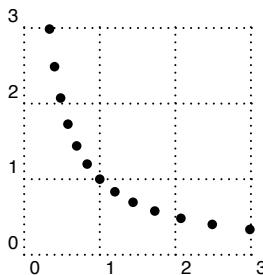
1 \psset{xunit=1.2pt}
2 \psplot[linecolor=gray,linewidth=1.5pt,plotstyle=curve]%
3   {0}{90}{x sin dup mul}
4 \psplot[plotpoints=100]{0}{90}{x sin x 2 div 2 exp cos mul}
5 \psline{<->}(0,-1)(0,1)
6 \psline{->}(100,0)

```

\parametricplot*[par]{t_{min}}{t_{max}}{function}

This is for a parametric plot of $(x(t), y(t))$. *function* is the PostScript code for calculating the pair $x(t)$ $y(t)$.

For example,



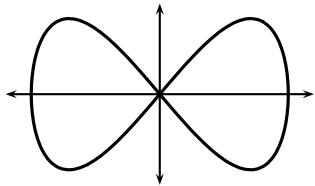
```

1 \parametricplot[plotstyle=dots,plotpoints=13]%
2 {-6}{6}{1.2 t exp 1.2 t neg exp}

```

plots 13 points from the hyperbola $xy = 1$, starting with $(1.2^{-6}, 1.2^6)$ and ending with $(1.2^6, 1.2^{-6})$.

Here is a parametric plot of $(\sin(t), \sin(2t))$:



```

1 \psset{xunit=1.7cm}
2 \parametricplot[linewidth=1.2pt,plotstyle=ccurve]%
3 {0}{360}{t sin t 2 mul sin}
4 \psline{<->}(0,-1.2)(0,1.2)
5 \psline{<->}(-1.2,0)(1.2,0)

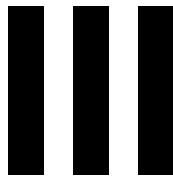
```

The number of points that the **\psplot** and **\parametricplot** commands calculate is set by the

plotpoints=int

Default: 50

parameter. Using **curve** or its variants instead of **line** and increasing the value of **plotpoints** are two ways to get a smoother curve. Both ways increase the imaging time. Which is better depends on the complexity of the computation. (Note that all PostScript lines are ultimately rendered as a series (perhaps short) line segments.) Mathematica generally uses **lineto** to connect the points in its plots. The default minimum number of plot points for Mathematica is 25, but unlike **\psplot** and **\parametricplot**, Mathematica increases the sampling frequency on sections of the curve with greater fluctuation.



More graphics parameters

The graphics parameters described in this part are common to all or most of the graphics objects.

12 Coordinate systems

The following manipulations of the coordinate system apply only to pure graphics objects.

A simple way to move the origin of the coordinate system to (x, y) is with the

origin={coor}

Default: Opt,Opt

This is the one time that coordinates *must* be enclosed in curly brackets {} rather than parentheses () .

A simple way to switch swap the axes is with the

swapaxes=true

Default: false

parameter. E.g., you might change your mind on the orientation of a plot after generating the data.

13 Line styles

The following graphics parameters (in addition to **linewidth** and **linecolor**) determine how the lines are drawn, whether they be open or closed curves.

linestyle=style

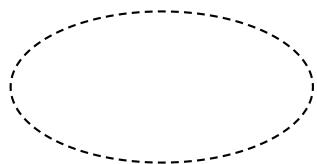
Default: solid

Valid styles are none, solid, dashed and dotted.

dash=dim1 dim2

Default: 5pt 3pt

The *black-white* dash pattern for the dashed line style. For example:



1 \psellipse[linestyle=dashed,dash=3pt 2pt](2,1)(2,1)

dotsep=dim**Default: 3pt**

The distance between dots in the dotted line style. For example



```
1 \psline[linestyle=dotted,dotsep=2pt]{l->}(4,1)
```

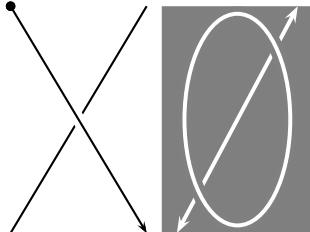
border=dim**Default: 0pt**

A positive value draws a border of width *dim* and color **bordercolor** on each side of the curve. This is useful for giving the impression that one line passes on top of another. The value is saved in the dimension register **\psborder**.

bordercolor=color**Default: white**

See **border** above.

For example:



```
1 \psline(0,0)(1.8,3)
2 \psline[border=2pt]{*->}(0,3)(1.8,0)
3 \psframe*[linecolor=gray](2,0)(4,3)
4 \psline[linecolor=white,linewidth=1.5pt]{<->}(2.2,0)(3.8,3)
5 \psellipse[linecolor=white,linewidth=1.5pt,
bordercolor=gray,border=2pt](3,1.5)(.7,1.4)
```

doubleline=true/false**Default: false**

When true, a double line is drawn, separated by a space that is **doublesep** wide and of color **doublecolor**. This doesn't work as expected with the dashed **linestyle**, and some arrows look funny as well.

doublesep=dim**Default: 1.25\pslinewidth**

See **doubleline**, above.

doublecolor=color**Default: white**

See **doubleline**, above.

Here is an example of double lines:



```
1 \psline[doubleline=true,linearc=.5,
2 doublesep=1.5pt]{<->}(0,0)(3,1)(4,0)
```

shadow=true/false**Default: false**

When true, a shadow is drawn, at a distance **shadowsize** from the original curve, in the direction **shadowangle**, and of color **shadowcolor**.

shadowsize=*dim* **Default: 3pt**

See **shadow**, above.

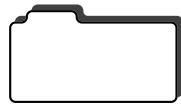
shadowangle=*angle* **Default: -45**

See **shadow**, above.

shadowcolor=*color* **Default: darkgray**

See **shadow**, above.

Here is an example of the **shadow** feature, which should look familiar:

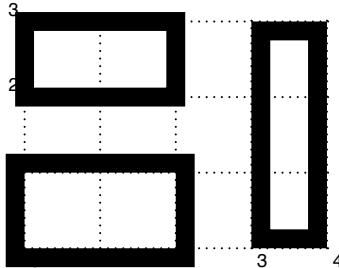


```
1 \pspolygon*[linearc=2pt,shadow=true,shadowangle=45,
2   xunit=1.1](-1,-.55)(-1,.5)(-.8,.55)(-.8,.65)
3   (-.2,.65)(-.2,.5)(1,.5)(1,-.55)
```

Here is another graphics parameter that is related to lines but that applies only to the closed graphics objects **\psframe**, **\pscircle**, **\psellipse** and **\pswedge**:

dimen=outer/inner/middle **Default: outer**

It determines whether the dimensions refer to the inside, outside or middle of the boundary. The difference is noticeable when the linewidth is large:



```
1 \psset{linewidth=.25cm}
2 \psframe[dimen=inner](0,0)(2,1)
3 \psframe[dimen=middle](0,2)(2,3)
4 \psframe[dimen=outer](3,0)(4,3)
```

With **\pswedge**, this only affects the radius; the origin always lies in the middle the boundary. The right setting of this parameter depends on how you want to align other objects.

14 Fill styles

The next group of graphics parameters determine how closed regions are filled. Even open curves can be filled; this does not affect how the curve is painted.

fillstyle=*style* **Default: none**

Valid styles are

none, solid, vlines, vlines*, hlines, hlines*, crosshatch, crosshatch* and boxfill.

vlines, hlines and crosshatch draw a pattern of lines, according to the four parameters list below that are prefixed with hatch. The * versions also fill the background, as in the solid style.

The boxfill style, which require to load the ‘pst-fill’ package, allow to fill (tile) the area with an arbitrary pattern defined using the `\psboxfill` macro (see the Part IX).

fillcolor=color

Default: white

The background color in the solid, vlines*, hlines* and crosshatch* styles.

hatchwidth=dim

Default: .8pt

Width of lines.

hatchsep=dim

Default: 4pt

Width of space between the lines.

hatchcolor=color

Default: black

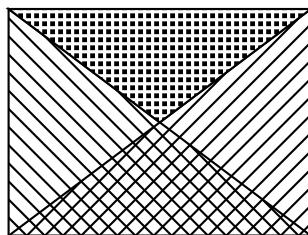
Color of lines. Saved in `\pshatchcolor`.

hatchangle=rot

Default: 45

Rotation of the lines, in degrees. For example, if **hatchangle** is set to 45, the vlines style draws lines that run NW-SE, and the hlines style draws lines that run SW-NE, and the crosshatch style draws both.

Here is an example of the vlines and related fill styles:⁶



```

1 \pspolygon[fillstyle=vlines](0,0)(0,3)(4,0)
2 \pspolygon[fillstyle=hlines](0,0)(4,3)(4,0)
3 \pspolygon[fillstyle=crosshatch*,fillcolor=black,
4   hatchcolor=white,hatchwidth=1.2pt,hatchsep=1.8pt,
5   hatchangle=0](0,3)(2,1.5)(4,3)

```

addfillstyle=style

Default:

This allow to use two different styles for the same area. This is specially useful when the boxfill style is used.

⁶PSTricks adjusts the lines relative to the resolution so that they all have the same width and the same intervening space. Otherwise, the checkered pattern in this example would be noticeably uneven, even on 300 dpi devices. This adjustment is resolution dependent, and may involve adjustments to the **hatchangle** when this is not initially a multiple of 45 degrees.

Each of the pure graphics objects (except those beginning with q) has a starred version that produces a solid object of color **linecolor**. (It automatically sets **linewidth** to zero, **fillcolor** to **linecolor**, **fillstyle** to solid, and **linestyle** to none.)

15 Arrowheads and such

Lines and other open curves can be terminated with various arrowheads, t-bars or circles. The

arrows=style

Default: -

parameter determines what you get. It can have the following values, which are pretty intuitive:⁷

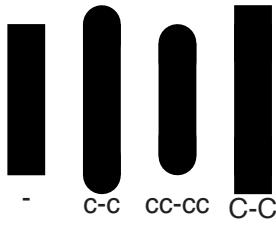
<i>Value</i>	<i>Example</i>	<i>Name</i>
-	—	None
<->	↔	Arrowheads.
><	→←	Reverse arrowheads.
<->>	↔→	Double arrowheads.
><->	→←↔	Double reverse arrowheads.
-	——	T-bars, flush to endpoints.
*- *	——	T-bars, centered on endpoints.
<->	←→	T-bars and arrowheads.
<*-> *	←→	T-bars and arrowheads, flush.
[-]	[—]	Square brackets.
(-)	(—)	Rounded brackets.
o-o	○—○	Circles, centered on endpoints.
-	●—●	Disks, centered on endpoints.
oo-oo	○—○	Circles, flush to endpoints.
-	●—●	Disks, flush to endpoints.
c-c	——	Extended, rounded ends.
cc-cc	——	Flush round ends.
C-C	——	Extended, square ends.

You can also mix and match. E.g., >, *-) and [-> are all valid values of the **arrows** parameter.

Well, perhaps the c, cc and C arrows are not so obvious. c and C correspond to setting PostScript's **linecap** to 1 and 2, respectively. cc is like c, but

⁷This is TeX's version of WYSIWYG.

adjusted so that the line flush to the endpoint. These arrows styles are noticeable when the **linewidth** is thick:



```

1 \psline[linewidth=.5cm](0,0)(0,2)
2 \psline[linewidth=.5cm]{c-c}(1,0)(1,2)
3 \psline[linewidth=.5cm]{cc-cc}(2,0)(2,2)
4 \psline[linewidth=.5cm]{C-C}(3,0)(3,2)

```

Almost all the open curves let you include the **arrows** parameters as an optional argument, enclosed in curly braces and before any other arguments (except the optional parameters argument). E.g., instead of

```
1 \psline[arrows=<,linestyle=dotted](3,4)
```

you can write

```
1 \psline[linestyle=dotted]{<}(3,4)
```

The exceptions are a few streamlined macros that do not support the use of arrows (these all begin with q).

The size of these line terminators is controlled by the following parameters. In the description of the parameters, the width always refers to the dimension perpendicular to the line, and length refers to a dimension in the direction of the line.

arrowsize=dim ‘num’

Default: 1.5pt 2

The width of arrowheads is *dim* plus *num* times **linewidth** (if the optional ‘num’ is included). See the diagram below.

arrowlength=num

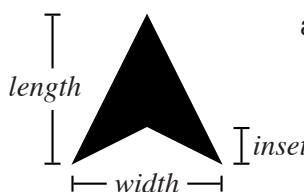
Default: 1.4

Length of arrowheads, as a fraction of the width, as shown below.

arrowinset=num

Default: .4

Size of inset for arrowheads, as a fraction of the length, as shown below.



$$\begin{aligned}
 \text{arrowsize} &= \text{dim num} \\
 \text{width} &= \text{num} \times \text{linewidth} + \text{dim} \\
 \text{length} &= \text{arrowlength} \times \text{width} \\
 \text{inset} &= \text{arrowinset} \times \text{length}
 \end{aligned}$$

tbarsize=dim ‘num’ Default: 2pt 5

The width of a t-bar, square bracket or rounded bracket is *dim* plus *num* times **linewidth** (if the optional ‘num’ is included). **linewidth**, plus *dim*.

bracketlength=num Default: .15

The height of a square bracket is *num* times its width.

rbracketlength=num Default: .15

The height of a round bracket is *num* times its width.

arrowscale=arrowscale=num1 num2 Default: 1

Imagine that arrows and such point down. This scales the width of the arrows by *num1* and the length (height) by *num2*. If you only include one number, the arrows are scaled the same in both directions. Changing **arrowscale** can give you special effects not possible by changing the parameters described above. E.g., you can change the width of lines used to draw brackets.

The size of dots is controlled by the **dotsize** parameter.

16 Custom styles

You can define customized versions of any macro that has parameter changes as an optional first argument using the **\newpsobject** command:

\newpsobject{name}{object}{par1=value1,... }

as in

```
1 \newpsobject{myline}{psline}{linecolor=green,linestyle=dotted}
2 \newpsobject{mygrid}{psgrid}{subgriddiv=1,griddots=10,
3   gridlabels=7pt}
```

The first argument is the name of the new command you want to define. The second argument is the name of the graphics object. Note that both of these arguments are given without the backslash. The third argument is the special parameter values that you want to set.

With the above examples, the commands **\myline** and **\mygrid** work just like the graphics object **\psline** it is based on, and you can even reset the parameters that you set when defining **\myline**, as in:

```
1 \myline[linecolor=gray,dotsep=2pt](5,6)
```

Another way to define custom graphics parameter configurations is with the

`\newpsstyle{name}{par1=value1,... }`

command. You can then set the **style** graphics parameter to *name*, rather than setting the parameters given in the second argument of `\newpsstyle`. For example,

```
1 \newpsstyle{mystyle}{linecolor=green,linestyle=dotted}
2 \psline[style=mystyle](5,6)
```

IV Custom graphics

17 The basics

PSTricks contains a large palette of graphics objects, but sometimes you need something special. For example, you might want to shade the region between two curves. The

\pscustom*[par]{*commands*}

command lets you “roll your own” graphics object.

Let’s review how PostScript handles graphics. A *path* is a line, in the mathematical sense rather than the visual sense. A path can have several disconnected segments, and it can be open or closed. PostScript has various operators for making paths. The end of the path is called the *current point*, but if there is no path then there is no current point. To turn the path into something visual, PostScript can *fill* the region enclosed by the path (that is what **fillstyle** and such are about), and *stroke* the path (that is what **linestyle** and such are about).

At the beginning of **\pscustom**, there is no path. There are various commands that you can use in **\pscustom** for drawing paths. Some of these (the open curves) can also draw arrows. **\pscustom** fills and strokes the path at the end, and for special effects, you can fill and stroke the path along the way using **\fill** and **\stroke** (see below).

Driver notes: **\pscustom** uses **\pstverb** and **\pstunit**. There are system-dependent limits on how long the argument of **\special** can be. You may run into this limit using **\pscustom** because all the PostScript code accumulated by **\pscustom** is the argument of a single **\special** command.

18 Parameters

You need to keep the separation between drawing, stroking and filling paths in mind when setting graphics parameters. The **linewidth** and **linecolor** parameters affect the drawing of arrows, but since the path commands do not stroke or fill the paths, these parameters, and the **linestyle**, **fillstyle** and related parameters, do not have any other effect (except that in some cases

linewidth is used in some calculations when drawing the path). **\pscustom** and **\fill** make use of **fillstyle** and related parameters, and **\pscustom** and **\stroke** make use of **linestyle** and related parameters.

For example, if you include

```
! \psline[linewidth=2pt, linecolor=blue, fillstyle=vlines]{<-}(3,3)(4,0)
```

in **\pscustom**, then the changes to **linewidth** and **linecolor** will affect the size and color of the arrow but not of the line when it is stroked, and the change to **fillstyle** will have no effect at all.

The **shadow**, **border**, **doubleline** and **showpoints** parameters are disabled in **\pscustom**, and the **origin** and **swapaxes** parameters only affect **\pscustom** itself, but there are commands (described below) that let you achieve these special effects.

The **dashed** and **dotted** line styles need to know something about the path in order to adjust the dash or dot pattern appropriately. You can give this information by setting the

linetype=int

Default: 0

parameter. If the path contains more than one disconnected segment, there is no appropriate way to adjust the dash or dot pattern, and you might as well leave the default value of **linetype**. Here are the values for simple paths:

<i>Value</i>	<i>Type of path</i>
0	Open curve without arrows.
-1	Open curve with an arrow at the beginning.
-2	Open curve with an arrow at the end.
-3	Open curve with an arrow at both ends.
1	Closed curve with no particular symmetry.
$n > 1$	Closed curve with n symmetric segments.

19 Graphics objects

You can use most of the graphics objects in **\pscustom**. These draw paths and making arrows, but do not fill and stroke the paths.

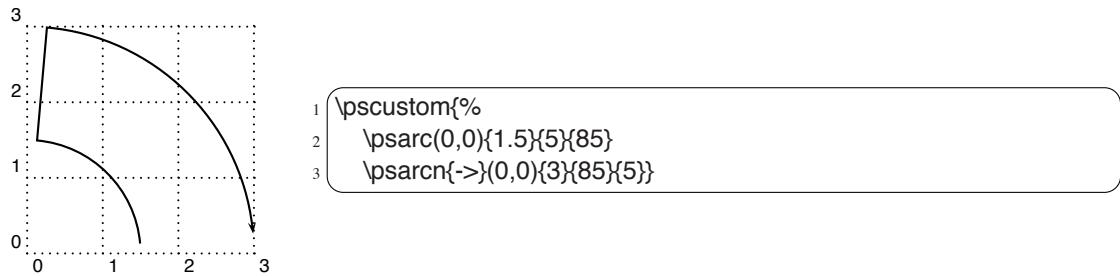
There are three types of graphics objects:

Special Special graphics objects include **\psgrid**, **\psdots**, **\qline** and **\qdisk**.
You cannot use special graphics objects in **\pscustom**.

Closed You are allowed to use closed graphics objects in **\pscustom**, but their effect is unpredictable.⁸ Usually you would use the open curves plus **\closepath** (see below) to draw closed curves.

Open The open graphics objects are the most useful commands for drawing paths with **\pscustom**. By piecing together several open curves, you can draw arbitrary paths. The rest of this section pertains to the open graphics objects.

By default, the open curves draw a straight line between the current point, if it exists, and the beginning of the curve, except when the curve begins with an arrow. For example



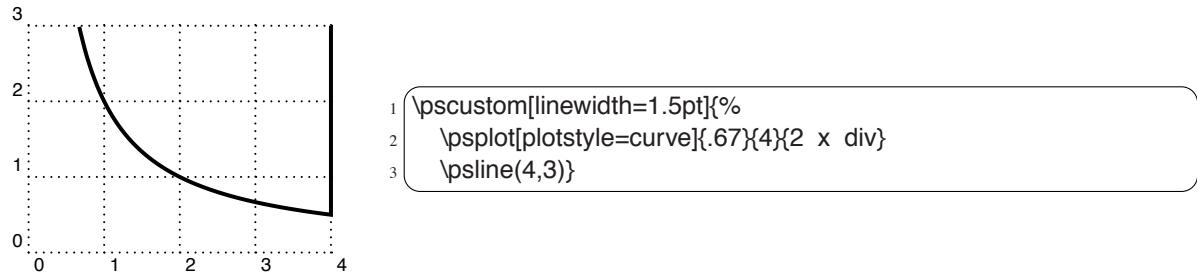
Also, the following curves make use of the current point, if it exists, as a first coordinate:

\psline and **\pscurve**.

The plot commands, with the **line** or **curve** **plotstyle**.

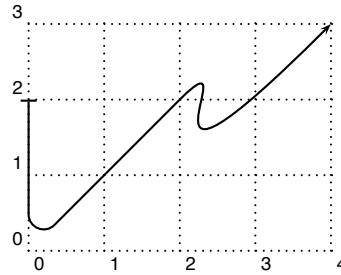
\psbezier if you only include three coordinates.

For example:



We'll see later how to make that one more interesting. Here is another example

⁸The closed objects never use the current point as an coordinate, but typically they will close any existing paths, and they might draw a line between the currentpoint and the closed curved.



```

1 \pscustom{%
2   \psline[linearc=.2]{l-}(0,2)(0,0)(2,2)
3   \psbezier{->}(2,2)(3,3)(1,0)(4,3)}

```

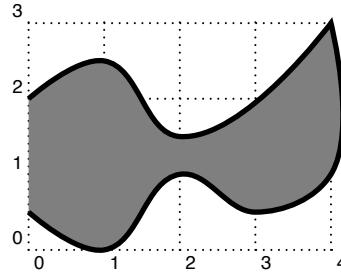
However, you can control how the open curves treat the current point with the

liftpen=0/1/2

Default: 0

parameter.

If **liftpen=0**, you get the default behavior described above. For example

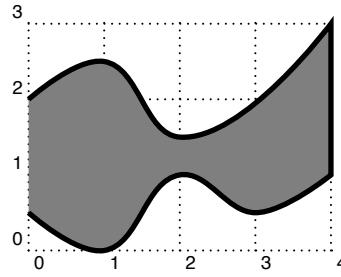


```

1 \pscustom[linewidth=2pt,fillstyle=solid,fillcolor=gray]{%
2   \pscurve(0,2)(1,2.5)(2,1.5)(4,3)
3   \pscurve(4,1)(3,0.5)(2,1)(1,0)(0,.5)}

```

If **liftpen=1**, the curves do not use the current point as the first coordinate (except **\psbezier**, but you can avoid this by explicitly including the first coordinate as an argument). For example:

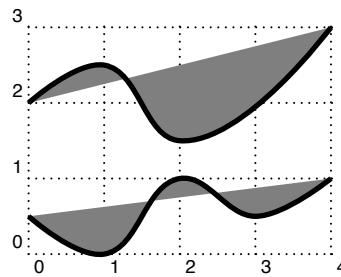


```

1 \pscustom[linewidth=2pt,fillstyle=solid,fillcolor=gray]{%
2   \pscurve(0,2)(1,2.5)(2,1.5)(4,3)
3   \pscurve[liftpen=1](4,1)(3,0.5)(2,1)(1,0)(0,.5)}

```

If **liftpen=2**, the curves do not use the current point as the first coordinate, and they do not draw a line between the current point and the beginning of the curve. For example



```

1 \pscustom[linewidth=2pt,fillstyle=solid,fillcolor=gray]{%
2   \pscurve(0,2)(1,2.5)(2,1.5)(4,3)
3   \pscurve[liftpen=2](4,1)(3,0.5)(2,1)(1,0)(0,.5)}

```

Later we will use the second example to fill the region between the two curves, and then draw the curves.

20 Safe tricks

The commands described under this heading, which can only be used in `\pscustom`, do not run a risk of PostScript errors (assuming your document compiles without TeX errors).

Let's start with some path, fill and stroke commands:

`\newpath`

Clear the path and the current point.

`\moveto(coor)`

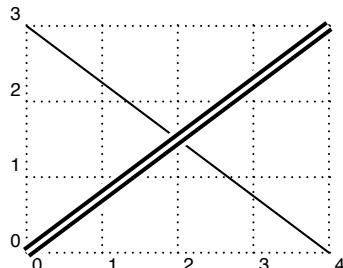
This moves the current point to (x, y) .

`\closepath`

This closes the path, joining the beginning and end of each piece (there may be more than one piece if you use `\moveto`).⁹

`\stroke[par]`

This strokes the path (non-destructively). `\pscustom` automatically strokes the path, but you might want to stroke it twice, e.g., to add a border. Here is an example that makes a double line and adds a border (this example is kept so simple that it doesn't need `\pscustom` at all):



```
1 \psline(0,3)(4,0)
2 \pscustom[linecolor=white,linewidth=1.5pt]{%
3   \psline(0,0)(4,3)
4   \stroke[linewidth=5\pslinewidth]
5   \stroke[linewidth=3\pslinewidth, linecolor=black]}
```

`\fill[par]`

This fills the region (non-destructively). `\pscustom` automatically fills the region as well.

`\gsave`

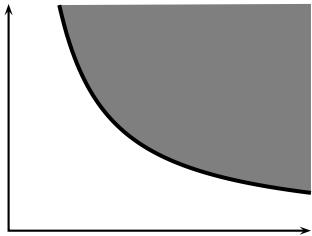
This saves the current graphics state (i.e., the path, color, line width, coordinate system, etc.) `\grestore` restores the graphics state. `\gsave` and `\grestore` must be used in pairs, properly nested with respect to TeX groups. You can have nested `\gsave-\grestore` pairs.

⁹Note that the path is automatically closed when the region is filled. Use `\closepath` if you also want to close the boundary.

\grestore

See above.

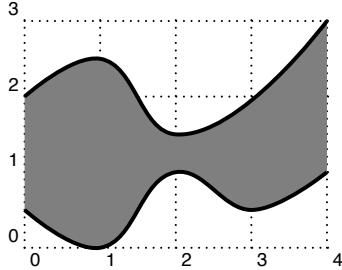
Here is an example that fixes an earlier example, using \gsave and \grestore:



```
1 \psline{<->}(0,3)(0,0)(4,0)
2 \pscustom[linewidth=1.5pt]{
3   \psplot[plotstyle=curve]{.67}{4}{2 x div}
4   \gsave
5     \psline(4,3)
6     \fill[fillstyle=solid,fillcolor=gray]
7   \grestore}
```

Observe how the line added by \psline(4,3) is never stroked, because it is nested in \gsave and \grestore.

Here is another example:



```
1 \pscustom[linewidth=1.5pt]{
2   \pscurve(0,2)(1,2.5)(2,1.5)(4,3)
3   \gsave
4     \pscurve[liftpen=1](4,1)(3,0.5)(2,1)(1,0)(0,.5)
5     \fill[fillstyle=solid,fillcolor=gray]
6   \grestore
7 \pscurve[linewidth=1.5pt](4,1)(3,0.5)(2,1)(1,0)(0,.5)
```

Note how I had to repeat the second \pscurve (I could have repeated it within \pscustom, with liftpen=2), because I wanted to draw a line between the two curves to enclose the region but I didn't want this line to be stroked.

The next set of commands modify the coordinate system.

\translate{coor}

Translate coordinate system by (x, y) . This shifts everything that comes later by (x, y) , but doesn't affect what has already been drawn.

\scale{num1 num2}

Scale the coordinate system in both directions by *num1*, or horizontally by *num1* and vertically by *num2*.

\rotate{angle}

Rotate the coordinate system by *angle*.

\swapaxes

Switch the x and y coordinates. This is equivalent to

```

1 \rotate{-90}
2 \scale{-1 1 scale}

```

\msave

Save the current coordinate system. You can then restore it with **\mrestore**. You can have nested **\msave-\mrestore** pairs. **\msave** and **\mrestore** do not have to be properly nested with respect to T_EX groups or **\gsave** and **\grestore**. However, remember that **\gsave** and **\grestore** also affect the coordinate system. **\msave-\mrestore** lets you change the coordinate system while drawing part of a path, and then restore the old coordinate system without destroying the path. **\gsave-\grestore**, on the other hand, affect the path and all other components of the graphics state.

\mrestore

See above.

And now here are a few shadow tricks:

\openshadow[par]

Strokes a replica of the current path, using the various shadow parameters.

\closedshadow[par]

Makes a shadow of the region enclosed by the current path as if it were opaque regions.

\movepath(coor)

Moves the path by (x, y) . Use **\gsave-\grestore** if you don't want to lose the original path.

21 Pretty safe tricks

The next group of commands are safe, *as long as there is a current point!*

\lineto(coor)

This is a quick version of **\psline(<coor>)**.

\rlineto(coor)

This is like **\lineto**, but (x, y) is interpreted relative to the current point.

\curveto(x1,y1)(x2,y2)(x3,y3)

This is a quick version of **\psbezier(x1,y1)(x2,y2)(x3,y3)**.

\rcurveto(x_1, y_1)(x_2, y_2)(x_3, y_3)

This is like **\curveto**, but (x_1, y_1) , (x_2, y_2) and (x_3, y_3) are interpreted relative to the current point.

22 For hackers only



For PostScript hackers, there are a few more commands. Be sure to read Appendix L.4 before using these. Needless to say:

Warning: Misuse of the commands in this section can cause PostScript errors.

The PostScript environment in effect with **\pscustom** has one unit equal to one \TeX pt .

\code{code}

Insert the raw PostScript code.

\dim{dim}

Convert the PSTricks dimension to the number of pt's, and inserts it in the PostScript code.

\coor(x_1, y_1)(x_2, y_2)...(x_n, y_n)

Convert one or more PSTricks coordinates to a pair of numbers (using pt units), and insert them in the PostScript code.

\rcoor(x_1, y_1)(x_2, y_2)...(x_n, y_n)

Like **\coor**, but insert the coordinates in reverse order.

\file{file}

This is like **\code**, but the raw PostScript is copied verbatim (except comments delimited by %) from *file*.

\arrows{arrows}

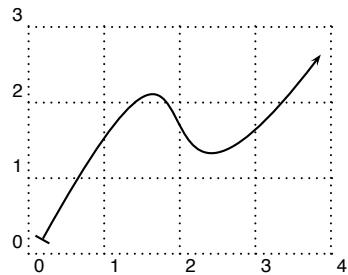
This defines the PostScript operators ArrowA and ArrowB so that

1	$x_2 \ y_2 \ x_1 \ y_1$	ArrowA
2	$x_2 \ y_2 \ x_1 \ y_1$	ArrowB

each draws an arrow(head) with the tip at (x_1, y_1) and pointing from (x_2, y_2) . ArrowA leaves the current point at end of the arrowhead, where a connect line should start, and leaves (x_2, y_2) on the stack. ArrowB does not change the current point, but leaves

1	$x_2 \ y_2 \ x_1' \ y_1'$
---	---------------------------

on the stack, where $(x1',y1')$ is the point where a connecting line should join. To give an idea of how this work, the following is roughly how PStricks draws a bezier curve with arrows at the end:



```
1 \pscustom{  
2   \arrows{l->}  
3   \code{  
4     80 140 5 5 ArrowA  
5     30 -30 110 75 ArrowB  
6     curveto}}
```

\setcolor{color}

Set the color to *color*.

V

Picture Tools

23 Pictures

The graphics objects and `\rput` and its variants do not change TeX's current point (i.e., they create a 0-dimensional box). If you string several of these together (and any other 0-dimensional objects), they share the same coordinate system, and so you can create a picture. For this reason, these macros are called *picture objects*.

If you create a picture this way, you will probably want to give the whole picture a certain size. You can do this by putting the picture objects in a `pspicture` environment,¹⁰ as in:

```
\pspicture*[baseline](x0,y0)(x1,y1)
  picture objects
\endpspicture
```

The picture objects are put in a box whose lower left-hand corner is at (x_0, y_0) (by default, $(0,0)$) and whose upper right-hand corner is at (x_1, y_1) .

By default, the baseline is set at the bottom of the box, but the optional argument [`<baseline>`] sets the baseline fraction `baseline` from the bottom. Thus, `baseline` is a number, generally but not necessarily between 0 and 1. If you include this argument but leave it empty ([]), then the baseline passes through the origin.

Normally, the picture objects can extend outside the boundaries of the box. However, if you include the `*`, anything outside the boundaries is clipped.

Besides picture objects, you can put anything in a `pspicture` that does not take up space. E.g., you can put in font declarations and use `\psset`, and you can put in braces for grouping. PSTricks will alert you if you include something that does take up space.¹¹

¹⁰LaTeX users can instead write:

```
\begin{pspicture} stuff \end{pspicture}
```

¹¹When PSTricks picture objects are included in a `pspicture` environment, they gobble up any spaces that follow, and any preceding spaces as well, making it less likely that extraneous space gets inserted. (PSTricks objects always ignore spaces that follow. If you also want them to try to neutralize preceding space when used outside the `pspicture` environment (e.g., in a LaTeX picture environment), then use the command `\KillGlue`. The command `\DontKillGlue` turns this behavior back off.)

You can use PSTricks picture objects in a \LaTeX picture environment, and you can use \LaTeX picture objects in a PSTricks **\pspicture** environment. However, the **\pspicture** environment makes \LaTeX 's picture environment obsolete, and has a few small advantages over the latter. Note that the arguments of the **\pspicture** environment work differently from the arguments of \LaTeX 's picture environment (i.e., the right way versus the wrong way).

Driver notes: The clipping option (*) uses **\pstVerb** and **\pstverbsscale** .

24 Placing and rotating whatever

PSTricks contains several commands for positioning and rotating an HR-mode argument. All of these commands end in **put** , and bear some similarity to \LaTeX 's **\put** command, but with additional capabilities. Like \LaTeX 's **\put** and unlike the box rotation macros described in Section 29, these commands do not take up any space. They can be used inside and outside **\pspicture** environments.

Most of the PSTricks **put** commands are of the form:

```
\put*{arg}{<rotation>}(<coor>){<stuff>}
```

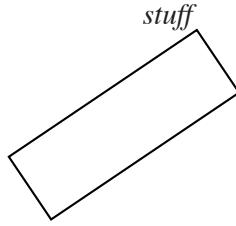
With the optional * argument, *stuff* is first put in a

```
1 \psframebox*[boxsep=false]{stuff}
```

thereby blotting out whatever is behind *stuff*. This is useful for positioning text on top of something else.

arg refers to other arguments that vary from one **put** command to another. The optional *rotation* is the angle by which *stuff* should be rotated; this argument works pretty much the same for all **put** commands and is described further below. The (*<coor>*) argument is the coordinate for positioning *stuff*, but what this really means is different for each **put** command. The (*<coor>*) argument is shown to be obligatory, but you can actually omit it if you include the *rotation* argument.

The *rotation* argument should be an angle, as described in Section 4, but the angle can be preceded by an *. This causes all the rotations (except the box rotations described in Section 29) within which the **\rput** command is be nested to be undone before setting the angle of rotation. This is mainly useful for getting a piece of text right side up when it is nested inside rotations. For example,



```

1 \rput{34}{%
2   \psframe(-1,0)(2,1)
3   \rput[br]{*0}(2,1){\emph{stuff}}}

```

There are also some letter abbreviations for the command angles. These indicate which way is up:

<i>Letter</i>	<i>Short for</i>	<i>Equiv. to</i>	<i>Letter</i>	<i>Short for</i>	<i>Equiv. to</i>
U	Up	0	N	North	*0
L	Left	90	W	West	*90
D	Down	180	S	South	*180
R	Right	270	E	East	*270

This section describes just two of the PSTricks put commands. The most basic one command is

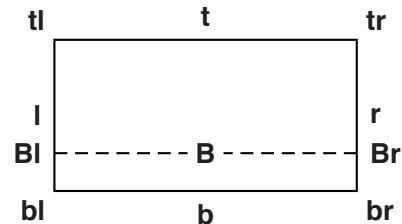
\rput*[refpoint]{rotation}{(x,y)}{stuff}

refpoint determines the reference point of *stuff*, and this reference point is translated to (x, y) .

By default, the reference point is the center of the box. This can be changed by including one or two of the following in the optional *refpoint* argument:

Horizontal		Vertical	
	Left	t	Top
r	Right	b	Bottom
		B	Baseline

Visually, here is where the reference point is set of the various combinations (the dashed line is the baseline):



There are numerous examples of **\rput** in this documentation, but for now here is a simple one:

```

1 \rput[b]{90}{-1,0}{Here is a marginal note.}

```

One common use of a macro such as `\rput` is to put labels on things. PSTricks has a variant of `\rput` that is especially designed for labels:

`\uput*{labelsep}[refangle]{rotation}{(x,y)}{stuff}`

This places *stuff* distance *labelsep* from (x, y) , in the direction *refangle*.

The default value of *labelsep* is the dimension register

`\pslabelsep`

You can also change this be setting the

`labelsep=dim`

Default: 5pt

parameter (but remember that `\uput` does have an optional argument for setting parameters).

Here is a simple example:

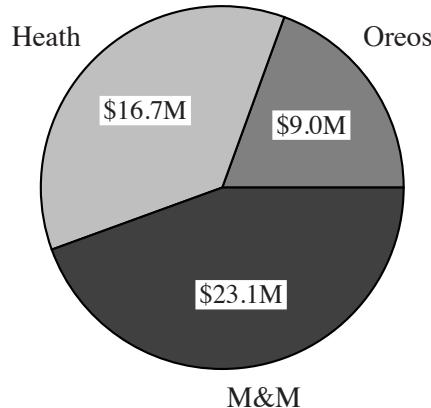


Here is a more interesting example where `\uput` is used to make a pie chart:

```

1 \psset{unit=1.2cm}
2 \pspicture(-2.2,-2.2)(2.2,2.2)
3   \pswedge[fillstyle=solid,fillcolor=gray]{2}{0}{70}
4   \pswedge[fillstyle=solid,fillcolor=lightgray]{2}{70}{200}
5   \pswedge[fillstyle=solid,fillcolor=darkgray]{2}{200}{360}
6   \SpecialCoor
7   \psset{framesep=1.5pt}
8   \rput(1.2;35){\psframebox*{\small\$9.0M}}
9   \uput{2.2}[45](0,0){Oreos}
10  \rput(1.2;135){\psframebox*{\small\$16.7M}}
11  \uput{2.2}[135](0,0){Heath}
12  \rput(1.2;280){\psframebox*{\small\$23.1M}}
13  \uput{2.2}[280](0,0){M&M}
14 \endpspicture

```



You can use the following abbreviations for *refangle*, which indicate the direction the angle points:¹²¹³

<i>Letter</i>	<i>Short for</i>	<i>Equiv. to</i>	<i>Letter</i>	<i>Short for</i>	<i>Equiv. to</i>
r	right	0	ur	up-right	45
u	up	90	ul	up-left	135
l	left	180	dl	down-left	225
d	down	270	dr	down-right	315

The first example could thus have been written:

```
(1,1)   1 \qdisk(1,1){1pt}
•      2 \uput[ur](1,1){(1,1)}
```

Driver notes: The rotation macros use **\pstVerb** and **\pstrotate**.

25 Repetition

The macro

¹²Using the abbreviations when applicable is more efficient.

¹³There is an obsolete command **\Rput** that has the same syntax as **\uput** and that works almost the same way, except the *refangle* argument has the syntax of **\uput**'s *refpoint* argument, and it gives the point in *stuff* that should be aligned with (x, y) . E.g.,

```
\qdisk(4,0){2pt} •
\Rput[tl](4,0){$(x,y)$} (x, y)
```

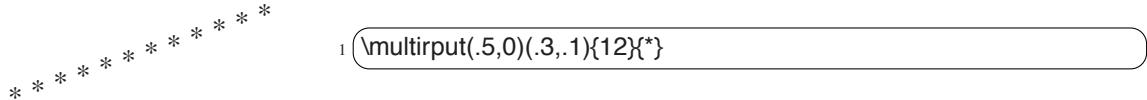
Here is the equivalence between **\uput**'s *refangle* abbreviations and **\Rput**'s *refpoint* abbreviations:

\uput	r	u	l	d	ur	ul	dr	dl
\Rput		b	r	t	bl	br	tr	rl

Some people prefer **\Rput**'s convention for specifying the position of *stuff* over **\uput**'s.

\multirput*[refpoint]{angle}{(x0,y0)}{(x1,y1)}{int}{stuff}

is a variant of **\rput** that puts down *int* copies, starting at (x_0, y_0) and advancing by (x_1, y_1) each time. (x_0, y_0) and (x_1, y_1) are always interpreted as Cartesian coordinates. For example:



If you want copies of pure graphics, it is more efficient to use

\multiputs{angle}{(x0,y0)}{(x1,y1)}{int}{graphics}

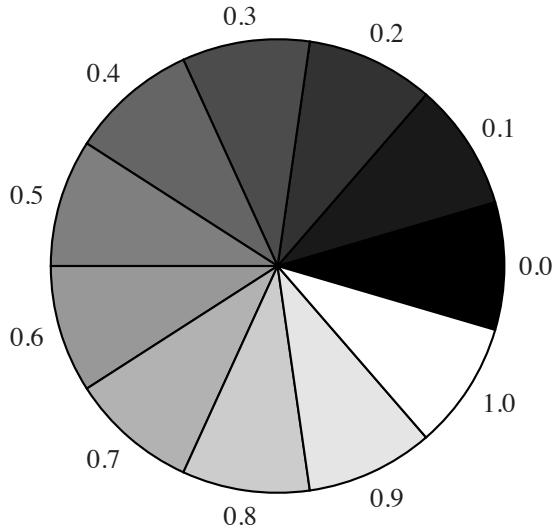
graphics can be one or more of the pure graphics objects described in Part II, or **\pscustom**. Note that **\multiputs** has the same syntax as **\multirput**, except that there is no *refpoint* argument (since the graphics are zero dimensional anyway). Also, unlike **\multirput**, the coordinates can be of any type. An Overfull **\hbox** warning indicates that the *graphics* argument contains extraneous output or space. For example:



PSTricks can heavily benefit of a much more general loop macro, called **\multido**. You must input the file `multido.tex` or `multido.sty`. See the documentation `multido.doc` for details. Here is a sample of what you can do:



```
1 \begin{pspicture}(-3.4,-3.4)(3.4,3.4)
2   \definecolor{mygray}{gray}{0}%
3   % Initialize 'mygray' for benefit
4   \psset{fillstyle=solid,fillcolor=mygray} %
5   % of this line.
6   \SpecialCoor
7   \degrees[1.1]
8   \multido{\n=0.0+0.1}{11}{%
9     \definecolor{mygray}{gray}{\n}%
10    \psset{fillcolor=mygray}%
11    \rput{\n}{\pswedge{3}{-.05}{.05}}%
12    \uput{3.2}{[\n]}(0,0){\small\n}%
13  }
```



All of these loop macros can be nested.

26 Axes

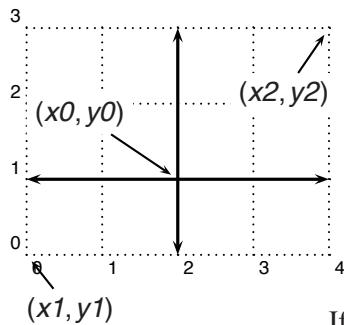


The axes command described in this section is defined in `pst-plot.tex` / `pst-plot.sty`, which you must input first. `pst-plot.tex`, in turn, will automatically input `multido.tex`, which is used for putting the labels on the axes.

The macro for making axes is:

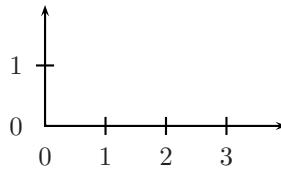
`\psaxes*[par]{arrows}(x0,y0)(x1,y1)(x2,y2)`

The coordinates must be Cartesian coordinates. They work the same way as with `\psgrid`. That is, if we imagine that the axes are enclosed in a rectangle, $(x1, y1)$ and $(x2, y2)$ are opposing corners of the rectangle. (I.e., the x-axis extends from $x1$ to $x2$ and the y-axis extends from $y1$ to $y2$.) The axes intersect at $(x0, y0)$. For example:



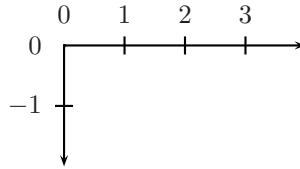
```
1 \psaxes[linewidth=1.2pt,labels=none,
2 ticks=none]{<->}(2,1)(0,0)(4,3)
```

If $(x0, y0)$ is omitted, then the origin is $(x1, y1)$. If both $(x0, y0)$ and $(x1, y1)$ are omitted, $(0,0)$ is used as the default. For example, when the axes enclose a single orthont, only $(x2, y2)$ is needed:



`\psaxes{->}(4,2)`

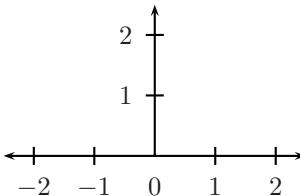
Labels (numbers) are put next to the axes, on the same side as $x1$ and $y1$. Thus, if we enclose a different orthont, the numbers end up in the right place:



`\psaxes{->}(4,-2)`

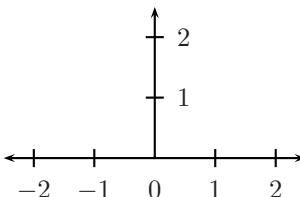
Also, if you set the **arrows** parameter, the first arrow is used for the tips at $x1$ and $y1$, while the second arrow is used for the tips at $x2$ and $y2$. Thus, in the preceding examples, the arrowheads ended up in the right place.¹⁴

When the axes don't just enclose an orthont, that is, when the origin is not at a corner, there is some discretion as to where the numbers should go. The rules for positioning the numbers and arrows described above still apply, and so you can position the numbers as you please by switching $y1$ and $y2$, or $x1$ and $x2$. For example, compare



`\psaxes{<->}(0,0)(-2.5,0)(2.5,2.5)`

with what we get when $x1$ and $x2$ are switched:



`\psaxes{<->}(0,0)(2.5,0)(-2.5,2.5)`

\psaxes puts the ticks and numbers on the axes at regular intervals, using the following parameters:

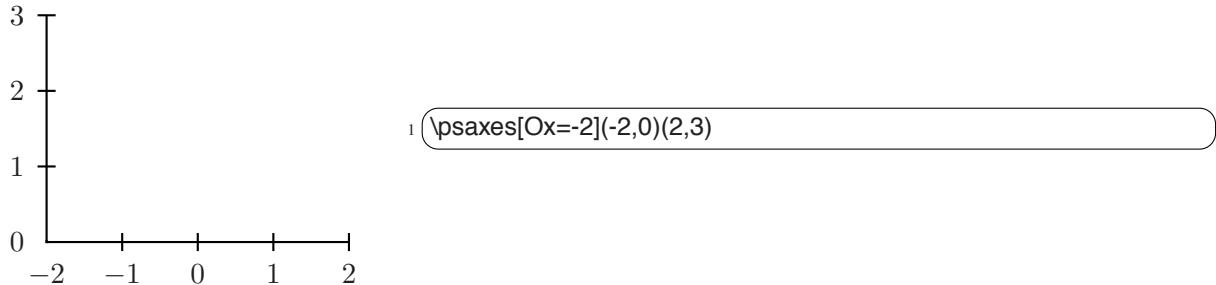
¹⁴Including a first arrow in these examples would have had no effect because arrows are never drawn at the origin.

<i>Horitontal</i>	<i>Vertical</i>	<i>Dflt</i>	<i>Description</i>
Ox=num	Oy=num	0	Label at origin.
Dx=num	Dy=num	1	Label increment.
dx=dim	dy=dim	Opt	Dist btwn labels.

When **dx** is 0, $Dx \setminus psxunit$ is used instead, and similarly for **dy**. Hence, the default values of Opt for **dx** and **dy** are not as peculiar as they seem.

You have to be very careful when setting **Ox**, **Dx**, **Oy** and **Dy** to non-integer values. `multido.tex` increments the labels using rudimentary fixed-point arithmetic, and it will come up with the wrong answer unless **Ox** and **Dx**, or **Oy** and **Dy**, have the same number of digits to the right of the decimal. The only exception is that **Ox** or **Oy** can always be an integer, even if **Dx** or **Dy** is not. (The converse does not work, however.)¹⁵

Note that **\psaxes**'s first coordinate argument determines the physical position of the origin, but it doesn't affect the label at the origin. E.g., if the origin is at (1,1), the origin is still labeled 0 along each axis, unless you explicitly change **Ox** and **Oy**. For example:



The ticks and labels use a few other parameters as well:

labels=all/x/y/none

Default: all

To specify whether labels appear on both axes, the x-axis, the y-axis, or neither.

showorigin=true/false

Default: true

If **true**, then labels are placed at the origin, as long as the label doesn't end up on one of the axes. If **false**, the labels are never placed at the origin.

ticks=all/x/y/none

Default: all

To specify whether ticks appear on both axes, the x-axis, the y-axis, or neither.

¹⁵For example, **Ox=1.0** and **Dx=1.4** is okay, as is **Ox=1** and **Dx=1.4**, but **Ox=1.4** and **Dx=1**, or **Ox=1.4** and **Dx=1.15**, is not okay. If you get this wrong, PStricks won't complain, but you won't get the right labels either.

tickstyle=full/top/bottom**Default: full**

For example, if **tickstyle=top**, then the ticks are only on the side of the axes away from the labels. If **tickstyle=bottom**, the ticks are on the same side as the labels. **full** gives ticks extending on both sides.

ticksizem=dim**Default: 3pt**

Ticks extend *dim* above and/or below the axis.

The distance between ticks and labels is **\pslabelsep**, which you can change with the **labelsep** parameter.

The labels are set in the current font (some of the examples above were preceded by **\small** so that the labels would be smaller). You can do fancy things with the labels by redefining the commands:

\pshlabel
\psvlabel

E.g., if you want change the font of the horizontal labels, but not the vertical labels, try something like

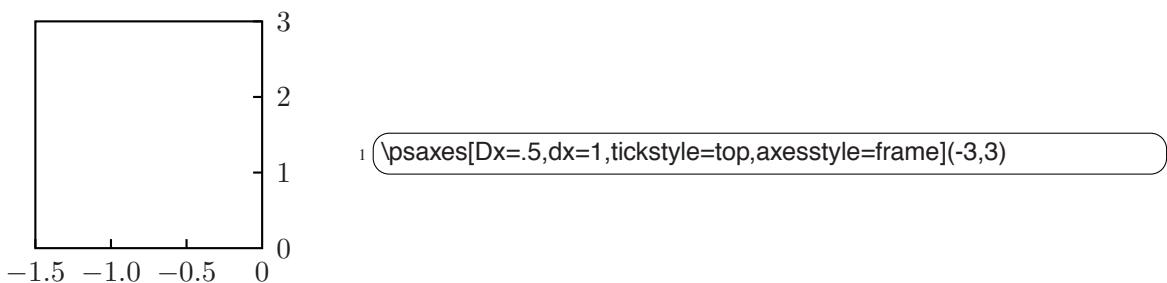
```
1 \def\pshlabel#1{\small #1}
```

You can choose to have a frame instead of axes, or no axes at all (but you still get the ticks and labels), with the parameter:

axesstyle=axes/frame/none**Default: axes**

The usual **linestyle**, **fillstyle** and related parameters apply.

For example:



The **\psaxes** macro is pretty flexible, but PStricks contains some other tools for making axes from scratch. E.g., you can use **\psline** and **\psframe** to draw axes and frames, respectively, **\multido** to generate labels (see the documentation for *multido.tex*), and **\multiput** to make ticks.

VI Text Tricks

27 Framed boxes

The macros for framing boxes take their argument, put it in an `\hbox`, and put a PostScript frame around it. (They are analogous to L^AT_EX's `\fbox`). Thus, they are composite objects rather than pure graphics objects. In addition to the graphics parameters for `\psframe`, these macros use the following parameters:

framesep=dim

Default: 3pt

Distance between each side of a frame and the enclosed box.

boxsep=true/false

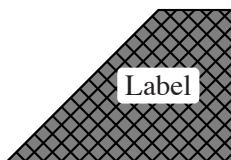
Default: true

When `true`, the box that is produced is the size of the frame or whatever that is drawn around the object. When `false`, the box that is produced is the size of whatever is inside, and so the frame is “transparent” to T_EX. This parameter only applies to `\psframebox`, `\pscirclebox`, and `\psovalbox`.

Here are the three box-framing macros:

`\psframebox*[par]{stuff}`

A simple frame (perhaps with rounded corners) is drawn using `\psframe`. The `*` option is of particular interest. It generates a solid frame whose color is `fillcolor` (rather than `linecolor`, as with the closed graphics objects). Recall that the default value of `fillcolor` is `white`, and so this has the effect of blotting out whatever is behind the box. For example,



```
1 \pspolygon[fillcolor=gray,fillstyle=crosshatch*](0,0)(3,0)(3,2)(2,2)
2 \rput(2,1){\psframebox*[framearc=.3]{Label}}
```

`\psdblframebox*[par]{stuff}`

This draws a double frame. It is just a variant of `\psframebox`, defined by

```
1 \newpsobject{psdblframebox}{psframebox}{doublesep=\pslinewidth}
```

For example,

```
1 \psdblframebox[linewidth=1.5pt]{%
2   \parbox[c]{6cm}{\raggedright A double frame is drawn
3   with the gap between lines equal to \texttt{doublesep}}}
```

A double frame is drawn with the gap
between lines equal to doublesep

\psshadowbox*[par]{stuff}

This draws a single frame, with a shadow.

Great Idea!!

```
1 \psshadowbox{\textbf{Great Idea!!}}
```

You can get the shadow with **\psframebox** just by setting the **shadowsize** parameter, but with **\psshadowbox** the dimensions of the box won't reflect the shadow (which may be what you want!).

\pscirclebox*[par]{stuff}

This draws a circle. With **boxsep=true**, the size of the box is close to but may be larger than the size of the circle. For example:



```
1 \pscirclebox{\begin{tabular}{c} You are \\ here \end{tabular}}
```

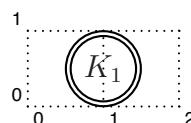
is distributed with

\cput*[par]{angle}(x,y){stuff}

This combines the functions of **\pscirclebox** and **\rput**. It is like

```
1 \rput{angle}(x0,y0){\pscirclebox*[par]{stuff}}
```

but it is more efficient. Unlike the **\rput** command, there is no argument for changing the reference point; it is always the center of the box. Instead, there is an optional argument for changing graphics parameters. For example



```
1 \cput[doubleline=true](1,.5){large $K_1$}
```

`\psovalbox*[par]{stuff}`

This draws an ellipse. If you want an oval with square sides and rounded corners, then use `\psframebox` with a positive value for `rectarc` or `linearc` (depending on whether `cornersize` is relative or absolute). Here is an example that uses `boxsep=false`:

At the introductory price of \$13.99, it pays to act now!

```
1 At the introductory price of
2 \psovalbox[boxsep=false, linecolor=darkgray]{\$13.99},
3 it pays to act now!
```

`\psdiabox*[par]{stuff}`

`\psdiabox` draws a diamond.



```
1 \psdiabox[shadow=true]{\Large\textbf{Happy?}}
```

`\pstriobox*[par]{stuff}`

`\pstriobox` draws a triangle.



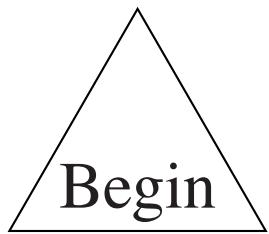
```
1 \pstriobox[trimode=R, framesep=5pt]{\Large\textbf{Begin}}
```

The triangle points in the direction:

`trimode=*`***U/D/R/L***

Default: U

If you include the optional `*`, then an equilateral triangle is drawn, otherwise, you get the minimum-area isosceles triangle.



```
1 \pstriobox[trimode=^U]{\Huge Begin}
```

You can define variants of these box framing macros using the `\newpsobject` command.

If you want to control the final size of the frame, independently of the material inside, nest `stuff` in something like L^AT_EX's `\makebox` command.

28 Clipping

The command

\clipbox[*dim*]{*stuff*}

puts *stuff* in an `\hbox` and then clips around the boundary of the box, at a distance *dim* from the box (the default is `0pt`).

The **\pspicture** environment also lets you clip the picture to the boundary.

The command¹⁶:

\psclip{*graphics*} ... {} \endpsclip

sets the clipping path to the path drawn by the *graphics* object(s), until the **\endpsclip** command is reached. **\psclip** and **\endpsclip** must be properly nested with respect to TeX grouping. Only pure graphics (those described in Part II and **\pscUSTOM**) are permitted. An Overfull `\hbox` warning indicates that the *graphics* argument contains extraneous output or space. Note that the *graphics* objects otherwise act as usual, and the **\psclip** does not otherwise affect the surrounded text. Here is an example:

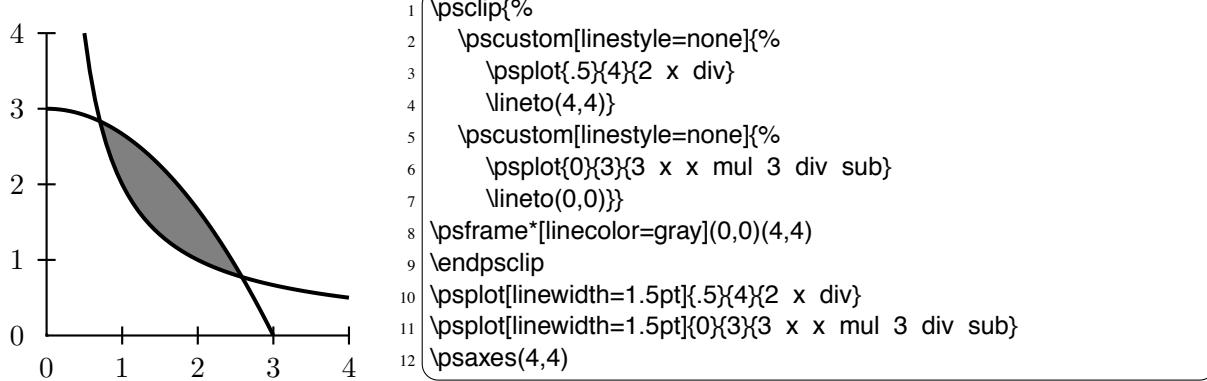
“One of the best new plays I have seen all year: cool, poetic, ironic ... ” proclaimed *The Guardian* upon the London premiere of “his extraordinary

```
1 \parbox{4.5cm}{%
2   \psclip{\psccurve*[linestyle=none](-3,-2)(0.3,-1.5)(2.3,-2)
3   (4.3,-1.5)(6.3,-2)(8,-1.5)(8,2)(-3,2)}
4   “One of the best new plays I have seen all year: cool,
5   poetic, ironic \ldots” proclaimed \emph{The Guardian} upon
6   the London premiere of this extraordinary play about a Czech
7   director and his actress wife, confronting exile in America.%
8   \vspace{-1cm}
9 }
```

If you don’t want the outline to be painted, you need to include **linestyle=none** in the parameter changes. You can actually include more than one graphics object in the argument, in which case the clipping path is set to the intersection of the paths.

\psclip can be a useful tool in picture environments. For example, here it is used to shade the region between two curves:

¹⁶LaTeX users can write `\begin{psclip} ... \end{psclip}` instead.



This is important to clearly understand this mechanism, so we give here the various steps of how it works in the preceding example:

```

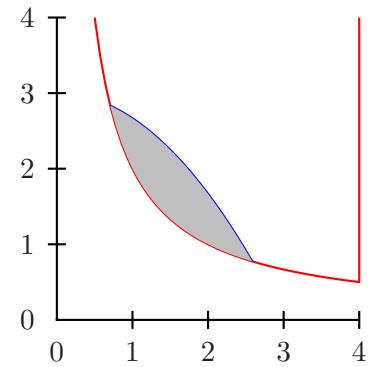
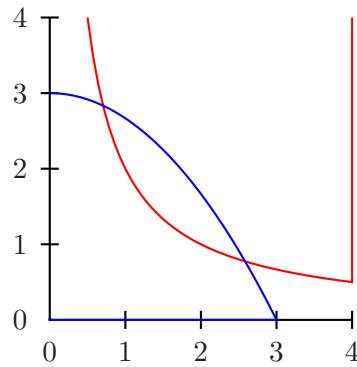
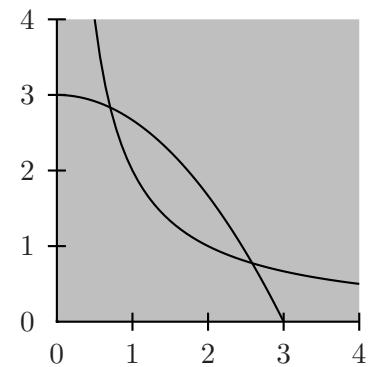
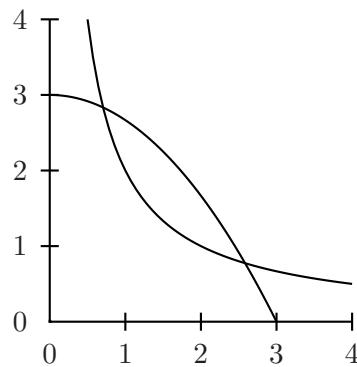
1 \pspicture(-0.5,-1)(4,4)
2   \psaxes(4,4)
3   \psplot{0.5}{4}{2 x div}
4   \psplot{0}{3}{3 x x mul 3 div sub}
5 \endpspicture
6 \hfill
7 \pspicture(-0.5,-1)(4,4)
8   \psframe*[linecolor=lightgray](4,4)
9   \psaxes(4,4)
10  \psplot{0}{3}{3 x x mul 3 div sub}
11  \psplot{0.5}{4}{2 x div}
12 \endpspicture
13
14 \pspicture(-0.5,-1)(4,4)
15   \psaxes(4,4)
16   \pscustom*[linecolor=red]{%
17     \psplot{0.5}{4}{2 x div}
18     \lineto(4,4)}
19   \pscustom*[linecolor=blue]{%
20     \psplot{0}{3}{3 x x mul 3 div sub}
21     \lineto(0,0)}
22 \endpspicture
23 \hfill
24 \pspicture(-0.5,-1)(4,4)
25   \psaxes(4,4)
26   \psclip{\pscustom*[linecolor=red]{%
27     \psplot{0.5}{4}{2 x div}
28     \lineto(4,4)}}
29     \pscustom*[linecolor=blue]{%
30       \psplot{0}{3}{3 x x mul 3 div sub}
31       \lineto(0,0)}}
32   \psframe*[linecolor=lightgray](4,4)
33 \endpsclip
34 \endpspicture
35
36 \pspicture(-0.5,-1)(4,4)

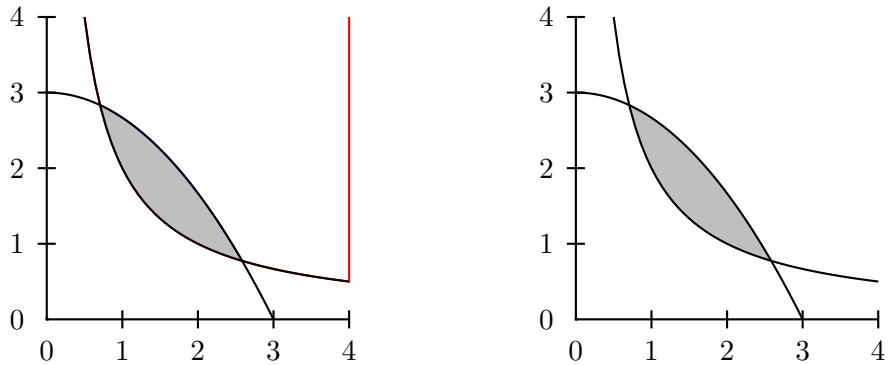
```

```

37  \psaxes(4,4)
38  \psclip{\pscustom[linecolor=red]{%
39      \psplot{0.5}{4}{2 x div}
40      \lineto(4,4)}}
41  \pscustom[linecolor=blue]{%
42      \psplot{0}{3}{3 x x mul 3 div sub}
43      \lineto(0,0)}}
44  \psframe*[linecolor=lightgray](4,4)
45  \endpsclip
46  \psplot{0}{3}{3 x x mul 3 div sub}
47  \psplot{0.5}{4}{2 x div}
48 \endpspicture
49 \hfill
50 \pspicture(-0.5,-1)(4,4)
51   \psaxes(4,4)
52   \psclip{\pscustom[linestyle=none]{%
53       \psplot{0.5}{4}{2 x div}
54       \lineto(4,4)}}
55   \pscustom[linestyle=none]{%
56       \psplot{0}{3}{3 x x mul 3 div sub}
57       \lineto(0,0)}}
58   \psframe*[linecolor=lightgray](4,4)
59 \endpsclip
60 \psplot{0}{3}{3 x x mul 3 div sub}
61 \psplot{0.5}{4}{2 x div}
62 \endpspicture

```





Driver notes: The clipping macros use `\pstverb{scale}` and `\pstVerb{Verb}`. Don't be surprised if PSTricks's clipping does not work or causes problem—it is never robust. `\endpsclip` uses `initclip`. This can interfere with other clipping operations, and especially if the `\TeX` document is converted to an Encapsulated PostScript file. The command `\AltClipMode` causes `\psclip` and `\endpsclip` to use `gsave` and `grestore` instead. This bothers some drivers, especially if `\psclip` and `\endpsclip` do not end up on the same page.

29 Rotation and scaling boxes

There are versions of the standard box rotation macros:

`\rotateleft{stuff}`
`\rotateright{stuff}`
`\rotatedown{stuff}`

stuff is put in an `\hbox` and then rotated or scaled, leaving the appropriate amount of spaces. Here are a few uninteresting examples:

Left Right Down

`\Large\bfseries\rotateleft{Left}\rotatedown{Down}\rotateright{Right}`

There are also two box scaling macros:

`\scalebox{num1 num2}{stuff}`

If you give two numbers in the first argument, *num1* is used to scale horizontally and *num2* is used to scale vertically. If you give just one number, the box is scaled by the same in both directions. You can't scale by zero, but negative numbers are OK, and have the effect of flipping the box around the axis. You never know when you need to do something like this (`\scalebox{-1 -1}{this}`).

`\scalebox{x,y}{stuff}`

This time, the first argument is a (Cartesian) coordinate, and the box is scaled to have width *x* and height (plus depth) *y*. If one of the dimensions is 0, the box is scaled by the same amount in both directions. For example:

Big and long

```
1 \scalebox{4,2}{Big and long}
```

PSTricks defines LR-box environments for all these box rotation and scaling commands:

```
1 \pslongbox{Rotateleft}{\rotateleft}
2 \pslongbox{Rotateright}{\rotateright}
3 \pslongbox{Rotatedown}{\rotatedown}
4 \pslongbox{Scalebox}{\scalebox}
5 \pslongbox{Scaleboxto}{\scaleboxto}
```

Here is an example where we **\Rotatedown** for the answers to exercises:

Question: How do
Democrats organize a
firing squad?
...
a circle,
Answer: First they get in

```
1 Question: How do Democrats organize a firing squad?
2
3 \begin{Rotatedown}
4   \parbox{\hsize}{Answer: First they get in a circle, \dots\hss}%
5 \end{Rotatedown}
```

See the documentation of the ‘fancybox’ package for tips on rotating a L^AT_EX table or figure environment, and other boxes.

VII

Nodes and Node Connections



All the commands described in this part are contained in the file `pst-node.tex` / `pst-node.sty`.

The node and node connection macros let you connect information and place labels, without knowing the exact position of what you are connecting or of where the lines should connect. These macros are useful for making graphs and trees, mathematical diagrams, linguistic syntax diagrams, and connecting ideas of any kind. They are the trickiest tricks in PSTRicks!

The node and node connection macros let you connect information and place labels, without knowing the exact position of what you are connecting or where the lines should connect. These macros are useful for making graphs and trees, mathematical diagrams, linguistic syntax diagrams, and connecting ideas of any kind. They are the trickiest tricks in PSTRicks!

There are three components to the node macros:

Node definitions The node definitions let you assign a name and shape to an object. See Section 30.

Node connections The node connections connect two nodes, identified by their names. See Section 31.

Node labels The node label commands let you affix labels to the node connections. See Section 32.

You can use these macros just about anywhere. The best way to position them depends on the application. For greatest flexibility, you can use the nodes in a `\pspicture`, positioning and rotating them with `\rput`. You can also use them in alignment environments. `pst-node.tex` contains a special alignment environment, `\psmatrix`, which is designed for positioning nodes in a grid, such as in mathematical diagrams and some graphs. `\psmatrix` is described in Section 35. `pst-node.tex` also contains high-level macros for trees. These are described in Part VIII.

But don't restrict yourself to these more obvious uses. For example:

I made the file symbol a node. Now I can draw an arrow so that you know what I am talking about.

```
1 \rnode{A}{%  
2   \parbox{4cm}{\raggedright  
3     I made the file symbol a node. Now I can draw an  
4       arrow so that you know what I am talking about.}}  
5 \ncarc[nodesep=8pt]{->}{A}{file}
```

30 Nodes

Nodes have a name, a boundary and a center.



Warning: The name is for referring to the node when making node connections and labels. You specify the name as an argument to the node commands. The name must contain only letters and numbers, and must begin with a letter. Bad node names can cause PostScript errors.

The center of a node is where node connections point to. The boundary is for determining where to connect a node connection. The various nodes differ in how they determine the center and boundary. They also differ in what kind of visible object they create.

Here are the nodes:

\rnode[refpoint]{name}{stuff}

\rnode puts *stuff* in a box. The center of the node is *refpoint*, which you can specify the same way as for **\rput**.

\Rnode*[par]{name}{stuff}

\Rnode also makes a box, but the center is set differently. If you align **\rnode**'s by their baseline, differences in the height and depth of the nodes can cause connecting lines to be not quite parallel, such as in the following example:

sp ————— Bit

```
1 \Large  
2 \rnode{A}{sp} \hskip 2cm \rnode{B}{Bit}  
3 \ncline{A}{B}
```

With **\Rnode**, the center is determined relative to the baseline:

sp ————— Bit

```
1 \Large  
2 \Rnode{A}{sp} \hskip 2cm \Rnode{B}{Bit}  
3 \ncline{A}{B}
```

You can usually get by without fiddling with the center of the node, but to modify it you set the

href=num
vref=dim

Default: 0
Default: .7ex

parameters. In the horizontal direction, the center is located fraction **href** from the center to the edge. E.g., if **href=-1**, the center is on the left edge of the box. In the vertical direction, the center is located distance **vref** from the baseline. The **vref** parameter is evaluated each

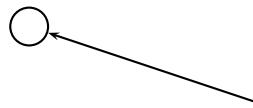
time **\Rnode** is used, so that you can use **ex** units to have the distance adjust itself to the size of the current font (but without being sensitive to differences in the size of letters within the current font).

\pnode(x,y){name}

This creates a zero dimensional node at (x,y) .

\cnode*[par](x,y){radius}{name}

This draws a circle. Here is an example with **\pnode** and **\cnode**:



```
1 \cnode(0,1){.25}{A}
2 \pnode(3,0){B}
3 \ncline{<-}{A}{B}
```

\Cnode*[par](x,y){name}

This is like **\cnode**, but the radius is the value of

radius=dim

Default: .25cm

This is convenient when you want many circle nodes of the same radius.

\circclenode*[par]{name}{stuff}

This is a variant of **\pscirclebox** that gives the node the shape of the circle.

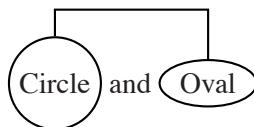
\cnodewithput*[par]{angle}(x,y){name}{stuff}

This is a variant of **\cput** that gives the node the shape of the circle. That is, it is like

```
1 \rput{angle}(x,y){\circclenode{name}{stuff}}
```

\ovalnode*[par]{name}{stuff}

This is a variant of **\psovalbox** that gives the node the shape of an ellipse. Here is an example with **\circclenode** and **\ovalnode**:



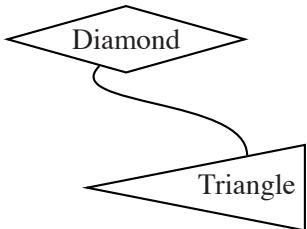
```
1 \circclenode{A}{Circle} and \ovalnode{B}{Oval}
2 \ncbar[angle=90]{A}{B}
```

\dianode*[par]{name}{stuff}

This is like **\diabox**.

\trinode*[par]{name}{stuff}

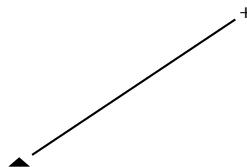
This is like **\tribox**.



```
1 \rput[tl](0,3){\dianode{A}{Diamond}}
2 \rput[br](4,0){\trinode[trimode=L]{B}{Triangle}}
3 \nccurve[angleA=-135,angleB=90]{A}{B}
```

\dotnode*[*par*](*x*,*y*)*{name}*

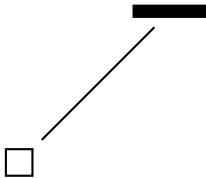
This is a variant of `\psdot`. For example:



```
1 \dotnode[dotstyle=triangle*,dotscale=2 1](0,0){A}
2 \dotnode[dotstyle=+](3,2){B}
3 \ncline[nodesep=3pt]{A}{B}
```

\fnode*[*par*](*x*,*y*)\{\i{name}\}

The `f` stands for “frame”. This is like, but easier than, putting a `\ps-frame` in an `\rnode`.



```
1 \fnode{A}  
2 \fnode*[framesize=1 5pt](2,2){B}  
3 \ncline[nodesep=3pt]{A}{B}
```

There are two differences between `\fnode` and `\psframe`:

- There is a single (optional) coordinate argument, that gives the *center* of the frame.
 - The width and height of the frame are set by the
framesize=dim1 ‘dim2’ **Default: 10pt**
parameter. If you omit *dim2*, you get a square frame.

31 Node connections

All the node connection commands begin with `nc`, and they all have the same syntax:^{17,18}

```
1 \nodeconnection[par]{arrows}{nodeA}{nodeB}
```

¹⁷The node connections can be used with `\pscustom`. The beginning of the node connection is attached to the current point by a straight line, as with `\psarc`.

¹⁸ See page 145 if you want to use the nodes as coordinates in other PSTricks macros.

A line of some sort is drawn from *nodeA* to *nodeB*. Some of the node connection commands are a little confusing, but with a little experimentation you will figure them out, and you will be amazed at the things you can do. When we refer to the A and B nodes below, we are referring only to the order in which the names are given as arguments to the node connection macros.¹⁹

The node connections use many of the usual graphics parameters, plus a few special ones. Let's start with one that applies to all the node connections:

nodesep=dim

Default: Opt

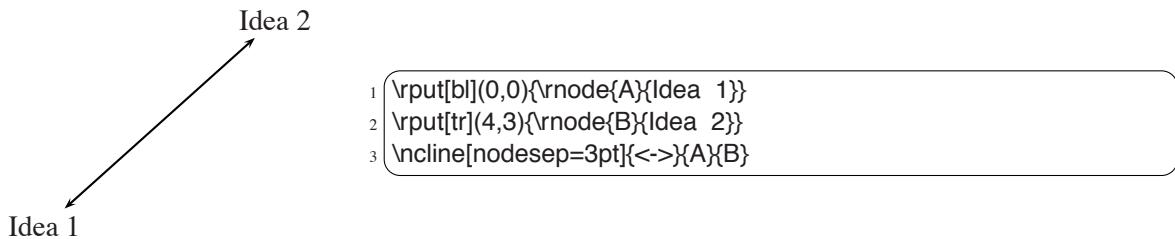
nodesep is the border around the nodes that is added for the purpose of determining where to connect the lines.

For this and other node connection parameters, you can set different values for the two ends of the node connection. Set the parameter **nodesepA** for the first node, and set **nodesepB** for the second node.

The first two node connections draw a line or arc directly between the two nodes:

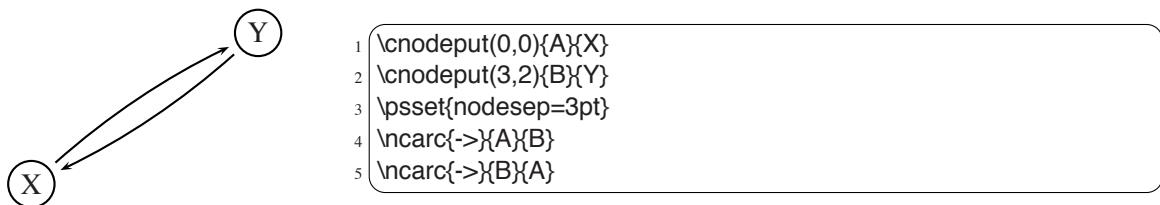
\ncline*[par]{arrows}{nodeA}{nodeB}

This draws a straight line between the nodes. For example:



\ncarc*[par]{arrows}{nodeA}{nodeB}

This connects the two nodes with an arc.



¹⁹When a node name cannot be found on the same page as the node connection command, you get either no node connection or a nonsense node connection. However, Te_X will not report any errors.

The angle between the arc and the line between the two nodes is²⁰

arcangle=angle

Default: 8

\ncline and **\ncarc** both determine the angle at which the node connections join by the relative position of the two nodes. With the next group of node connections, you specify one or both of the angles in absolute terms, by setting the

angle=angle

Default: 0

(and **angleA** and **angleB**) parameter.

You also specify the length of the line segment where the node connection joins at one or both of the ends (the “arms”) by setting the

arm=dim

Default: 10pt

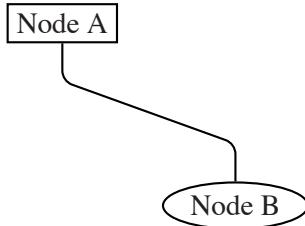
(and **armA** and **armB**) parameter.

These node connections all consist of several line segments, including the arms. The value of **linearc** is used for rounding the corners.

Here they are, starting with the simplest one:

\ncdiag*[par]{arrows}{nodeA}{nodeB}

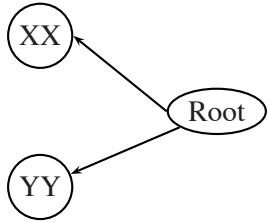
An arm is drawn at each node, joining at angle **angleA** or **angleB**, and with a length of **armA** or **armB**. Then the two arms are connected by a straight line, so that the whole line has three line segments. For example:



```
1 \rput[tl](0,3){\rnode{A}\psframebox{Node A}}
2 \rput[br](4,0){\ovalnode{B}(Node B)}
3 \ncdiag[angleA=-90, angleB=90, arm=.5, linearc=.2]{A}{B}
```

You can also set one or both of the arms to zero length. For example, if you set **arm=0**, the nodes are connected by a straight line, but you get to determine where the line connects (whereas the connection point is determined automatically by **\ncline**). Compare this use of **\ncdiag** with **\ncline** in the following example:

²⁰Rather than using a true arc, **\ncarc** actually draws a bezier curve. When connecting two circular nodes using the default parameter values, the curve will be indistinguishable from a true arc. However, **\ncarc** is more flexible than an arc, and works right connecting nodes of different shapes and sizes. You can set **arcangleA** and **arcangleB** separately, and you can control the curvature with the **ncurv** parameter, which is described on page ??.



```

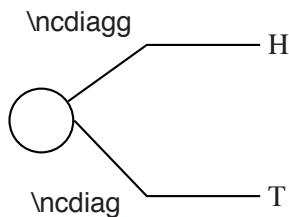
1 \rput[r](4,1){ovalnode{R}{Root}}
2 \cnodeput(1,2){A}{XX}
3 \cnodeput(1,0){B}{YY}
4 \ncdiag[angleB=180, arm=0]<-(A)(R)
5 \ncline{<--}(B)(R)

```

(Note that in this example, the default value **angleA=0** is used.)

\ncdiagg*[par]{arrows}{nodeA}{nodeB}

\ncdiagg is similar to **\ncdiag**, but only the arm for node A is drawn. The end of this arm is then connected directly to node B. Compare **\ncdiagg** with **\ncdiag** when **armB=0**:



```

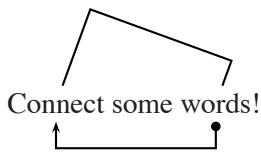
1 \cnode(0,0){12pt}{a}
2 \rput[l](3,1){\rnode{b}{H}}
3 \rput[l](3,-1){\rnode{c}{T}}
4 \ncdiagg[angleA=180, armA=1.5, nodesepA=3pt]{b}{a}
5 \ncdiag[angleA=180, armA=1.5, armB=0, nodesepA=3pt]{c}{a}

```

You can use **\ncdiagg** with **armA=0** if you want a straight line that joins to node A at the angle you specify, and to node B at an angle that is determined automatically.

\ncbar*[par]{arrows}{nodeA}{nodeB}

This node connection consists of a line with arms dropping “down”, at right angles, to meet two nodes at an angle **angleA**. Each arm is at least of length **armA** or **armB**, but one may be need to be longer.



```

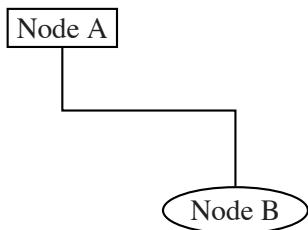
1 \rnode{A}{Connect some \rnode{B}{words}!}
2 \ncbar[nodesep=3pt,angle=-90]<-*{A}{B}
3 \ncbar[nodesep=3pt,angle=70]{A}{B}

```

Generally, the whole line has three straight segments.

\ncangle*[par]{arrows}{nodeA}{nodeB}

Now we get to a more complicated node connection. **\ncangle** typically draws three line segments, like **\ncdiag**. However, rather than fixing the length of arm A, we adjust arm A so that the line joining the two arms meets arm A at a right angle. For example:

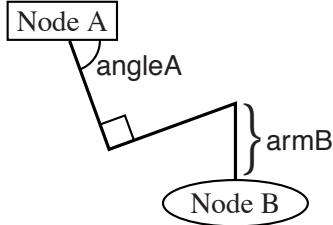


```

1 \rput[tl](0,3){\rnode{A}{\psframebox{Node A}}}
2 \rput[br](4,0){ovalnode{B}{Node B}}
3 \ncangle[angleA=-90,angleB=90,armB=1cm]{A}{B}

```

Now watch what happens when we change **angleA**:

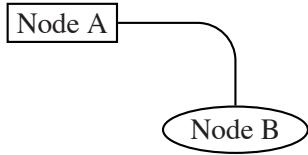


```

1 \rput[tl](0,3){\rnode{A}{\psframebox{Node A}}}
2 \rput[br](4,0){\ovalnode{B}{Node B}}
3 \ncangle[angleA=-70,angleB=90,armB=1cm,linewidth=1.2pt]{A}{B}

```

\ncangle is also a good way to join nodes by a right angle, with just two line segments, as in this example:



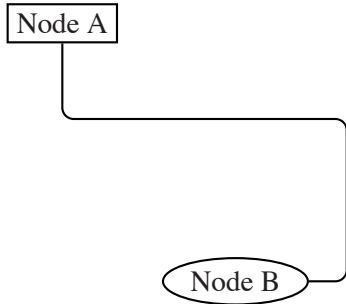
```

1 \rput[tl](0,2){\rnode{A}{\psframebox{Node A}}}
2 \rput[br](4,0){\ovalnode{B}{Node B}}
3 \ncangle[angleB=90, armB=0, linearC=.5]{A}{B}

```

\ncangles*[par]{arrows}{nodeA}{nodeB}

\ncangles is similar to **\ncangle**, but the length of arm A is fixed by the **armA** parameter. Arm A is connected to arm B by two line segments that meet arm A and each other at right angles. The angle at which they join arm B, and the length of the connecting segments, depends on the positions of the two arms. **\ncangles** generally draws a total of four line segments.²¹ For example:



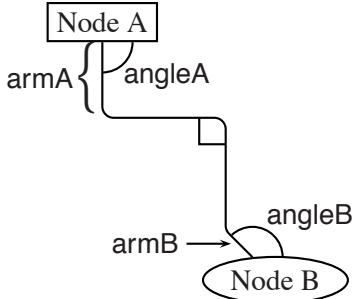
```

1 \rput[tl](0,4){\rnode{A}{\psframebox{Node A}}}
2 \rput[br](4,0){\ovalnode{B}{Node B}}
3 \ncangles[angleA=-90, armA=1cm, armB=.5cm, linearC=.15]{A}{B}

```

Let's see what happens to the previous example when we change **angleB**:

²¹Hence there is one more angle than **\ncangle**, and hence the **s** in **\ncangles**.



```

1 \rput[tl](0,4){\rnode{A}{\psframebox{Node A}}}
2 \rput[br](4,0){\ovalnode{B}{Node B}}
3 \ncangles[angleA=-90, angleB=135, armA=1cm, armB=.5cm,
4           linearc=.15]{A}{B}

```

\ncloloop*[par]{arrows}{nodeA}{nodeB}

\ncloloop is also in the same family as **\ncangle** and **\ncangles**, but now typically 5 line segments are drawn. Hence, **\ncloloop** can reach around to opposite sides of the nodes. The lengths of the arms are fixed by **armA** and **armB**. Starting at arm A, **\ncloloop** makes a 90 degree turn to the left, drawing a segment of length

loopsize=dim

Default: 1cm

This segment connects to arm B the way arm A connects to arm B with **\ncline**; that is, two more segments are drawn, which join the first segment and each other at right angles, and then join arm B. For example:



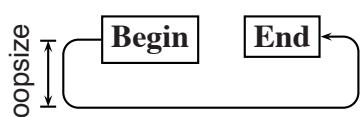
```

1 \rnode{a}{\psframebox{Huge A loop}}
2 \ncloloop[angleB=180,loopsize=1,arm=.5,linearc=.2]{->}{a}{a}

```

In this example, node A and node B are the same node! You can do this with all the node connections (but it doesn't always make sense).

Here is an example where **\ncloloop** connects two different nodes:



```

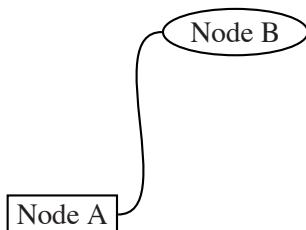
1 \parbox{3cm}{%
2 \rnode{A}{\psframebox{\large\textbf{Begin}}}
3 \vspace{1cm}\hspace*{fill}
4 \rnode{B}{\psframebox{\large\textbf{End}}}
5 \ncloloop[angleA=180,loopsize=.9,arm=.5,linearc=.2]{->}{A}{B}
}

```

The next two node connections are a little different from the rest.

\nccurve*[par]{arrows}{nodeA}{nodeB}

\nccurve draws a bezier curve between the nodes.



```

1 \rput[bl](0,0){\rnode{A}{\psframebox{Node A}}}
2 \rput[tr](4,3){\ovalnode{B}{Node B}}
3 \nccurve[angleB=180]{A}{B}

```

You specify the angle at which the curve joins the nodes by setting the **angle** (and **angleA** and **angleB**) parameter. The distance to the control points is set with the

ncurv=num

Default: .67

(and **ncurvA** and **ncurvB**) parameter. A lower number gives a tighter curve. (The distance between the beginning of the arc and the first control point is one-half **ncurvA** times the distance between the two endpoints.)

\nccircle*[par]{arrows}{node}{radius}

\nccircle draws a circle, or part of a circle, that, if complete, would pass through the center of the node counterclockwise, at an angle of **angleA**.



```

1 \rnode{A}\textbf{back}
2 \nccircle[nodesep=3pt]{>}{A}{.7cm}
3 \kern 5pt

```

\nccircle can only connect a node to itself; it is the only node connection with this property. **\nccircle** is also special because it has an additional argument, for specifying the radius of the circle.

The last two node connections are also special. Rather than connecting the nodes with an open curve, they enclose the nodes in a box or curved box. You can think of them as variants of **\ncline** and **\ncarc**. In both cases, the half the width of the box is

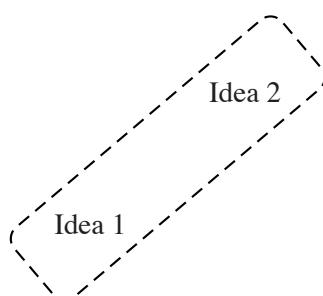
boxsize=dim

Default: .4cm

You have to set this yourself to the right size, so that the nodes fit inside the box. The **boxsize** parameter actually sets the **boxheight** and **boxdepth** parameters. The ends of the boxes extend beyond the nodes by **nodedsepA** and **nodedsepB**.

\ncbox*[par]{nodeA}{nodeB}

\ncbox encloses the nodes in a box with straight sides. For example:



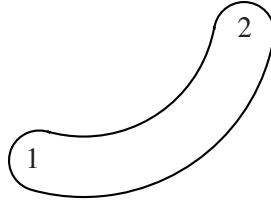
```

1 \rput[bl](.5,0){\rnode{A}{Idea 1}}
2 \rput[tr](3.5,2){\rnode{B}{Idea 2}}
3 \ncbox[nodesep=.5cm,boxsize=.6,linear=2,
4 linestyle=dashed]{A}{B}

```

`\ncarcbox*[par]{nodeA}{nodeB}`

`\ncarcbox` encloses the nodes in a curved box that is **arcangleA** away from the line connecting the two nodes.



```

1 \rput[bl](.5,0){\rnode{A}{1}}
2 \rput[tr](3.5,2){\rnode{B}{2}}
3 \ncarcbox[nodesep=.2cm,boxsize=.4,linearc=.4,
4 arcangle=50]{<->}{A}{B}

```

The arc is drawn counterclockwise from node A to node B.

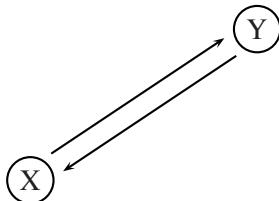
There is one other node connection parameter that applies to all the node connections, except `\ncarcbox`:

offset=dim

Default: Opt

(You can also set `offsetA` and `offsetB` independently.) This shifts the point where the connection joins up by *dim* (given the convention that connections go from left to right).

There are two main uses for this parameter. First, it lets you make two parallel lines with `\ncline`, as in the following example:



```

1 \cnodeput(0,0){A}{X}
2 \cnodeput(3,2){B}{Y}
3 \psset{nodesep=3pt,offset=4pt,arrows=->}
4 \ncline{A}{B}
5 \ncline{B}{A}

```

Second, it lets you join a node connection to a rectangular node at a right angle, without limiting yourself to positions that lie directly above, below, or to either side of the center of the node. This is useful, for example, if you are making several connections to the same node, as in the following example:

Word1 and Word2 and Word3

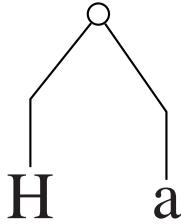


```

1 \rnode{A}{Word1} and \rnode{B}{Word2} and \rnode{C}{Word3}
2 \ncbar[offsetB=4pt,angleA=-90,nodesep=3pt]{->}{A}{B}
3 \ncbar[offsetA=4pt,angleA=-90,nodesep=3pt]{->}{B}{C}

```

Sometimes you might be aligning several nodes, such as in a tree, and you want to ends or the arms of the node connections to line up. This won't happen naturally if the nodes are of different size, as you can see in this example:

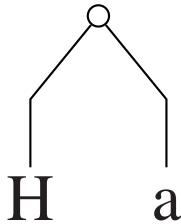


```

1 \Huge
2 \cnode(1,3){4pt}{a}
3 \rput[B](0,0){\Rnode{b}{H}}
4 \rput[B](2,0){\Rnode{c}{a}}
5 \psset{angleA=90,armA=1,nodesepA=3pt}
6 \ncdiag{b}{a}
7 \ncdiag{c}{a}

```

If you set the **nodesep** or **arm** parameter to a negative value, PSTricks will measure the distance to the beginning of the node connection or to the end of the arm relative to the center of the node, rather than relative to the boundary of the node or the beginning of the arm. Here is how we fix the previous example:



```

1 \Huge
2 \cnode(1,3){4pt}{a}
3 \rput[B](0,0){\Rnode{b}{H}}
4 \rput[B](2,0){\Rnode{c}{a}}
5 \psset{angleA=90,armA=1,YnodesepA=12pt}
6 \ncdiag{b}{a}
7 \ncdiag{c}{a}

```

Note also the use of **\Rnode**.

One more parameter trick: By using the **border** parameter, you can create the impression that one node connection passes over another.

The node connection commands make interesting drawing tools as well, as an alternative to **\psline** for connecting two points. There are variants of the node connection commands for this purpose. Each begins with **pc** (for “point connection”) rather than **nc**. E.g.,

```
1 \pcarc{<->}(3,4)(6,9)
```

gives the same result as

```

1 \pnode(3,4){A}
2 \pnode(6,9){B}
3 \pcarc{<->}{A}{B}

```

Only **\nccircle** does not have a **pc** variant:

<i>Command</i>	<i>Corresponds to:</i>
<code>\pcline{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\ncline</code>
<code>\pccurve{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\nccurve</code>
<code>\pcarc{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\ncarc</code>
<code>\pcbar{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\ncbar</code>
<code>\pcdiag{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\ncdiag</code>
<code>\pcdiagg{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\ncdiagg</code>
<code>\pcangle{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\ncangle</code>
<code>\pcangles{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\ncangles</code>
<code>\pcloop{arrows}{x_1, y_1}(x_2, y_2)</code>	<code>\ncloop</code>
<code>\pcbox(x_1, y_1)(x_2, y_2)</code>	<code>\ncbox</code>
<code>\pcarcbox(x_1, y_1)(x_2, y_2)</code>	<code>\ncarcbox</code>

32 Node connections labels: I

Now we come to the commands for attaching labels to the node connections. The label command must come right after the node connection to which the label is to be attached. You can attach more than one label to a node connection, and a label can include more nodes.

The node label commands must end up on the same T_EX page as the node connection to which the label corresponds.

There are two groups of connection labels, which differ in how they select the point on the node connection. In this section we describe the first group:

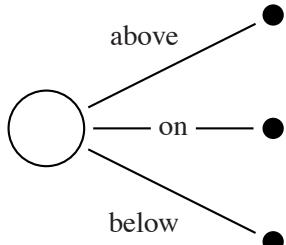
```
\ncput*[par]{stuff}
\naput*[par]{stuff}
\bput*[par]{stuff}
```

These three command differ in where the labels end up with respect to the line:

\ncput	<i>on the line</i>
\naput	<i>above the line</i>
\bput	<i>below the line</i>

(using the convention that node connections go from left to right).

Here is an example:



```

1 \cnode(0,0){.5cm}{root}
2 \cnode*(3,1.5){4pt}{A}
3 \cnode*(3,0){4pt}{B}
4 \cnode*(3,-1.5){4pt}{C}
5 \psset{nodesep=3pt}
6 \ncline{root}{A}
7 \naput{above}
8 \ncline{root}{B}
9 \ncput*[on]
10 \ncline{root}{C}
11 \nbput{below}

```

\naput and **\nbput** use the same algorithm as **\rput** for displacing the labels, and the distance between the line and labels is **labelsep** (at least if the lines are straight).

\ncput uses the same system as **\rput** for setting the reference point. You change the reference point by setting the

ref=ref

Default: c

parameter.

Rotation is also controlled by a graphics parameter:

nrot=rot

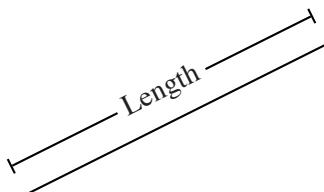
Default: 0

rot can be in any of the forms suitable for **\rput**, and you can also use the form

```
1 {:angle}
```

The angle is then measured with respect to the node connection. E.g., if the angle is **{:U}**, then the label runs parallel to the node connection. Since the label can include other put commands, you really have a lot of control over the label position.

The next example illustrates the use **{:<angle>}**, the **offset** parameter, and **\pcline**:

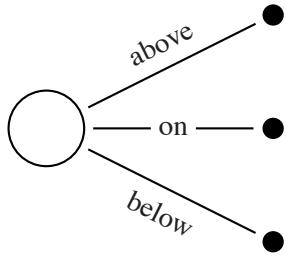


```

1 \pspolygon(0,0)(4,2)(4,0)
2 \pcline[offset=12pt]{|-l|}(0,0)(4,2)
3 \ncput*[nrot=:U]{Length}

```

Here is a repeat of an earlier example, now using **{:<angle>}**:



```

1 \cnode(0,0){.5cm}{root}
2 \cnode*(3,1.5){4pt}{A}
3 \cnode*(3,0){4pt}{B}
4 \cnode*(3,-1.5){4pt}{C}
5 \psset{nodesep=3pt,nrot=:U}
6 \ncline{root}{A}
7 \nput{above}
8 \ncline{root}{B}
9 \nput*[on]
10 \ncline{root}{C}
11 \nbput{below}

```

The position on the node connection is set by the

npos=num

Default:

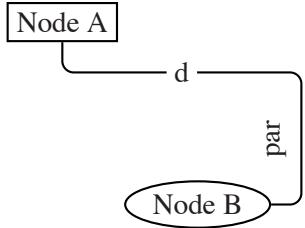
parameter, roughly according to the following scheme: Each node connection has potentially one or more segments, including the arms and connecting lines. A number **npos** between 0 and 1 picks a point on the first segment from node A to B (fraction **npos** from the beginning to the end of the segment), a number between 1 and 2 picks a number on the second segment, and so on.

Each node connection has its own default value of **npos**. If you leave the **npos** parameter value empty (e.g., [npos=]), then the default is substituted. This is the default mode.

Here are the details for each node connection:

<i>Connection</i>	<i>Segments</i>	<i>Range</i>	<i>Default</i>
\ncline	1	$0 \leq pos \leq 1$	0.5
\nccurve	1	$0 \leq pos \leq 1$	0.5
\ncarc	1	$0 \leq pos \leq 1$	0.5
\ncbar	3	$0 \leq pos \leq 3$	1.5
\ncdiag	3	$0 \leq pos \leq 3$	1.5
\ncdiagg	2	$0 \leq pos \leq 2$	0.5
\ncangle	3	$0 \leq pos \leq 3$	1.5
\ncangles	4	$0 \leq pos \leq 4$	1.5
\ncloop	5	$0 \leq pos \leq 5$	2.5
\ccircle	1	$0 \leq pos \leq 1$	0.5
\ncbox	4	$0 \leq pos \leq 4$	0.5
\ncarcbox	4	$0 \leq pos \leq 4$	0.5

Here is an example:

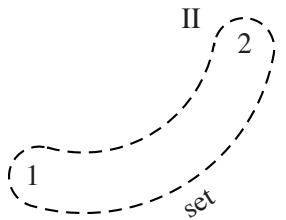


```

1 \rput[tl](0,3){\rnode{A}{\psframebox{Node A}}}
2 \rput[br](3.5,0){\ovalnode{B}{Node B}}
3 \ncangles[angleA=-90,arm=.4cm,linearc=.15]{A}{B}
4 \nccut*[d]
5 \nbput[nrot=:D,npos=2.5]{par}

```

With **\ncbox** and **\ncarcbox**, the segments run counterclockwise, starting with the lower side of the box. Hence, with **\nbput** the label ends up outside the box, and with **\naput** the label ends up inside the box.



```

1 \rput[bl](.5,0){\rnode{A}{1}}
2 \rput[tr](3.5,2){\rnode{B}{2}}
3 \ncarcbox[nodesep=.2cm,boxsize=.4,linearc=.4,
           arcangle=50,linestyle=dashed]{<->}{A}{B}
4 \nbput[nrot=:U]{set}
5 \nbput[npos=2]{II}

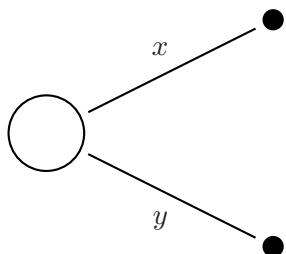
```

If you set the parameter

shortput=nab/tabl/r/tab

Default: none

to nab, then immediately following a node connection or another node connection label you can use **^** instead of **\naput** and **_** instead of **\nbput**.



```

1 \cnode(0,0){.5cm}{root}
2 \cnode*(3,1.5){4pt}{A}
3 \cnode*(3,-1.5){4pt}{C}
4 \psset{nodesep=3pt,shortput=nab}
5 \ncline{root}{A}^{x}
6 \ncline{root}{C}_

```

You can still have parameter changes with the short **^** and **_** forms. Another example is given on page 78.

If you have set **shortput=nab**, and then you want to use a true **^** or **_** character right after a node connection, you must precede the **^** or **_** by **{}** so that PSTricks does not convert it to **\naput** or **\nbput**.

You can change the characters that you use for the short form with the

\MakeShortNab{char1}{char2}

command.²²

²²You can also use **\MakeShortNab** if you want to use **^** and **_** with non-standard category codes. Just invoke the command after you have made your **\catcode** changes.

The **shortput=tabl** and **shortput=tab** options are described on pages 75 and ??, respectively.

33 Node connection labels: II

Now the second group of node connections:

```
\tput*[par]{stuff}
\tlput*[par]{stuff}
\trput*[par]{stuff}
\thput*[par]{stuff}
\taput*[par]{stuff}
\tbput*[par]{stuff}
```

The difference between these commands and the \n*put commands is that these find the position as an intermediate point between the centers of the nodes, either in the horizontal or vertical direction. These are good for trees and mathematical diagrams, where it can sometimes be nice to have the labels be horizontally or vertically aligned. The t stands for “tree”.

You specify the position by setting the

tpos=num	Default: .5
-----------------	--------------------

parameter.

\tput, **\lput** and **\rput** find the position that lies fraction *tpos* in the *vertical* direction from the upper node to the lower node. **\hput**, **\aput** and **\bput** find the position that lies fraction *tpos* in the *horizontal* direction from the left node to the right node. Then the commands put the label on or next to the line, as follows:

Command	Direction	Placement
\tput	vertical	middle
\lput	vertical	left
\rput	vertical	right
\hput	horizontal	middle
\aput	horizontal	above
\bput	horizontal	below

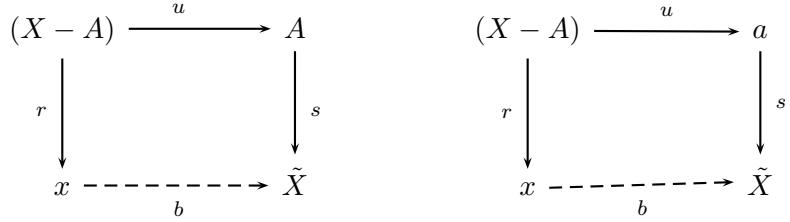
Here is an example:

```
1 \[
2   \setlength{\arraycolsep}{1.1cm}
3   \begin{array}{cc}
4     \Rnode{a}{(X-A)} & \Rnode{b}{(A)} \\\[1.5cm]
```

```

5   \Rnode{c}{x} & \Rnode{d}{\tilde{X}}\\
6   \end{array}\\
7   \psset{nodesep=5pt,arrows=>}\\
8   \everypsbox{\scriptstyle}\\
9   \ncline{a}{c}\tput{r}\\
10  \ncline{a}{b}\tput{u}\\
11  \ncline[linestyle=dashed]{c}{d}\tput{b}\\
12  \ncline{b}{d}\tput{s}\\
13 \]

```



On the left is the diagram with `\tput`, `\rput`, `\tbput` and `\Rnode`, as shown in the code. On the right is the same diagram, but with `\nput` `\nbput` and `\rnnode`.

These do not have a rotation argument or parameter. However, you can rotate *stuff* in 90 degree increments using box rotations (e.g., `\rotateleft`).

If you set `shortput=tabl`, then you can use the following single-character abbreviations for the t put commands:

Char. *Short for:*

<code>^</code>	<code>\tput</code>
<code>_</code>	<code>\tbput</code>
<code><</code>	<code>\lput</code>
<code>></code>	<code>\rput</code>

You can change the character abbreviations with

`\MakeShortTablr{char1}{char2}{char3}{char4}`

The t put commands, including an example of `shortput=tabl`, will be shown further when we get to mathematical diagrams and trees.

Driver notes: The node macros use `\pstVerb` and `\pstverbsscale`.

34 Attaching labels to nodes

The command

`\nput*[par]{refangle}{name}{stuff}`

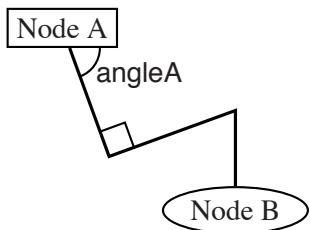
affixes *stuff* to node *name*. It is positioned distance **labelsep** from the node, in the direction *refangle* from the center of the node. The algorithm is the same as for **\uput**. If you want to rotate the node, set the

rot=*rot*

Default: 0

parameter, where *rot* is a rotation that would be valid for `\rput`.²³ The position of the label also takes into account the **offsetA** parameter. If **labelsep** is negative, then the distance is from the center of the node rather than from the boundary, as with **nodesep**.

Here is how I used `\input` to mark an angle in a previous example:



```

1 \rput[br](4,0){\ovalnode{B}{Node B}}
2 \rput[tl](0,3){\rnode{A}{\psframebox{Node A}}}
3 \nput[labelsep=0]{-70}{A}%
4   \psarcn(0,0){.4cm}{0}{-70}
5   \uput{.4cm}{-35}(0,0){\texttt{angleA}}}
6 \ncangle[angleA=-70,angleB=90,armB=1cm,linewidth=1.2pt]{A}{B}
7 \nccput[nrot=:U,npos=1]{\psframe[dimen=middle](0,0)(.35,.35)}

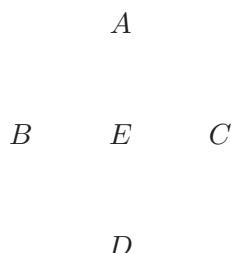
```

35 Mathematical diagrams and graphs

For some applications, such as mathematical diagrams and graphs, it is useful to arrange nodes on a grid. You can do this with alignment environments, such as **TEX**'s `\halign` primitive, **LATEX**'s `tabular` environment, and **AMS-TEX**'s `\matrix`, but **PSTricks** contains its own alignment environment that is especially adapted for this purpose²⁴

\psmatrix{} ... \endpsmatrix

Here is an example



```

1 $ 
2 \psmatrix[colsep=1cm,rowsep=1cm]
3   & A \\
4   B & E & C \\
5   & D &
6 \endpsmatrix
7 $

```

As an alignment environment, `\psmatrix` is similar to AMS-T_EX's `\matrix`. There is no argument for specifying the columns. Instead, you can just use

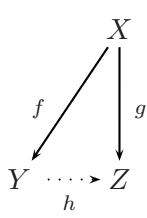
²³Not to be confused with the `nput` parameter.

²⁴LAT_EX users can instead write:

```
\begin{psmatrix} stuff \end{psmatrix}
```

as many columns as you need. The entries are horizontally centered. Rows are ended by `\end{psmatrix}` can be used in or out of math mode.

Our first example wasn't very interesting, because we didn't make use of the nodes. Actually, each entry is a node. The name of the node in row `row` and column `col` is `{<row>,<col>}`, with no spaces. Let's see some node connections:

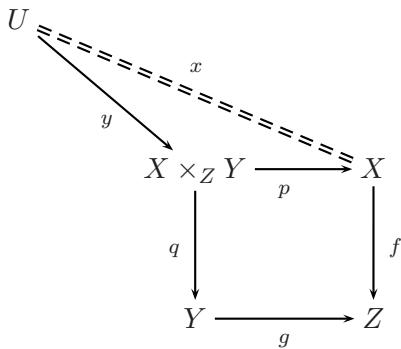


```

1 $ 
2 \psmatrix[colsep=1cm]
3   & X \\
4   Y & Z
5 \endpsmatrix
6 \everypsbox{\scriptstyle}%
7 \psset{nodesep=3pt,arrows=>}
8 \ncline{1,2}{2,1}
9 \tput{f}
10 \ncline{1,2}{2,2}
11 \tput{g}
12 \ncline[linestyle=dotted]{2,1}{2,2}
13 \bput{h}
14 $

```

You can include the node connections inside the `\psmatrix`, in the last entry and right before `\end{psmatrix}`. One advantage to doing this is that `shortput=tab` is the default within a `\psmatrix`.



```

1 $ 
2 \begin{psmatrix}
3   U & \\
4   & X\times_Z Y & X \\
5   & & & Y & Z
6 \psset{arrows=>,nodesep=3pt}
7 \everypsbox{\scriptstyle}
8 \ncline{1,1}{2,2}_y
9 \ncline[doubleline=true,linestyle=dashed]{2,2}{3,2}_x
10 \ncline{2,2}{3,2}_q
11 \ncline{2,2}{2,3}_p
12 \ncline{2,3}{3,3}_f
13 \ncline{3,2}{3,3}_g
14 \end{psmatrix}
15 $

```

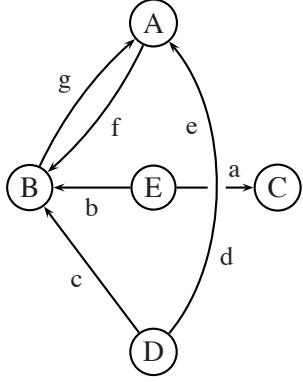
You can change the kind of nodes that are made by setting the

mnode=*type*

Default: R

parameter. Valid types are `R`, `r`, `C`, `f`, `p`, `circle`, `oval`, `dia`, `tri`, `dot` and `none`, standing for `\Rnode`, `\rnode`, `\Cnode`, `\fnode`, `\pnode`, `\circlenode`, `\ovalnode`, `\dotnode` and no node, respectively. Note that for circles, you use `mnode=C` and set the radius with the `radius` parameter.

For example:



```

1 \psmatrix[mnode=circle,colsep=1]
2   & A &
3   B & E & C \\
4   & D &
5 \endpsmatrix
6 \psset{shortput=nab,arrows=->,labelsep=3pt}
7 \small
8 \ncline{2,2}{2,3}^{\tiny [npos=.75]}{a}
9 \ncline{2,2}{2,1}^{\tiny [b]}
10 \ncline{3,2}{2,1}^{\tiny [c]}
11 \ncarc[arcangle=-40,border=3pt]{3,2}{1,2}
12 \tiny [npos=.3]{d}^{\tiny [npos=.7]}{e}
13 \ncarc[arcangle=12]{1,2}{2,1}^{\tiny [f]}
14 \ncarc[arcangle=12]{2,1}{1,2}^{\tiny [g]}

```

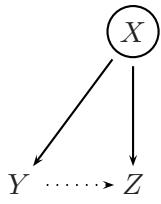
Note that a node is made only for the non-empty entries. You can also specify a node for the empty entries by setting the

emnode=type

Default: none

parameter.

You can change parameters for a single entry by starting this entry with the parameter changes, enclosed in square brackets. Note that the changes affect the way the node is made, but not contents of the entry (use **\psset** for this purpose). For example:



```

1 $ 
2 \psmatrix[colsep=1cm]
3   & [mnode=circle] X \\
4   Y & Z
5 \endpsmatrix
6 \psset{nodesep=3pt,arrows=->}
7 \ncline{1,2}{2,1}
8 \ncline{1,2}{2,2}
9 \ncline[linestyle=dotted]{2,1}{2,2}
10 $

```

If you want your entry to begin with a [that is not meant to indicate parameter changes, the precede it by {.

You can assign your own name to a node by setting the

name=name

Default:

parameter at the beginning of the entry, as described above. You can still refer to the node by {<row>,<col>}, but here are a few reasons for giving your own name to a node:

- The name may be easier to keep track of;
- Unlike the {`<row>`,`<col>`} names, the names you give remain valid even when you add extra rows or columns to your matrix.
- The names remain valid even when you start a new `\psmatrix` that reuses the {`<row>`,`<col>`} names.

Here a few more things you should know:

- The baselines of the nodes pass through the centers of the nodes. `\psmatrix` achieves this by setting the

`nodelign=true/false` **Default: false**

parameter to `true`. You can also set this parameter outside of `\psmatrix` when you want this kind of alignment.

- You can left or right-justify the nodes by setting the

`mcol=l/r/c` **Default: c**

parameter. `l`, `r` and `c` stand for left, right and center, respectively.

- The space between rows and columns is set by the

`rowsep=dim` **Default: 1.5cm**

`colsep=dim` **Default: 1.5cm**

parameters.

- If you want all the nodes to have a fixed width, set

`mnodesize=dim` **Default: -1pt**

to a positive value.

- If `\psmatrix` is used in math mode, all the entries are set in math mode, but you can switch a single entry out of math mode by starting and ending the entry with `$`.

- The radius of the `c mnode` (corresponding to `\cnode`) is set by the `radius` parameter.

- Like in L^AT_EX, you can end a row with `\[<dim>`] to insert an extra space `dim` between rows.

- The command `\psrowhookii` is executed, if defined, at the beginning of every entry in row ii (row 2), and the command `\pscolhookv` is executed at the beginning of every entry in column v (etc.). You can use these hooks, for example, to change the spacing between two columns, or to use a special `mnode` for all the entries in a particular row.

- An entry can itself be a node. You might do this if you want an entry to have two shapes.
- If you want an entry to stretch across several (*int*) columns, use the **\psspan{int}**

at the end of the entry. This is like PLAIN T_EX's **\multispan**, or L_AT_EX's **\multicolumn**, but the template for the current column (the first column that is spanned) is still used. If you want wipe out the template as well, use **\multispan<int>** *at the beginning of the entry* instead. If you just want to wipe out the template, use **\omit** before the entry.

- **\psmatrix** can be nested, but then all node connections and other references to the nodes in the {<row>,<col>} form for the nested matrix *must go inside* the **\psmatrix**. This is how PSTricks decides which matrix you are referring to. It is still neatest to put all the node connections towards the end; just be sure to put them before **\endpsmatrix**. Be careful also not to refer to a node until it actually appears. The whole matrix can itself go inside a node, and node connections can be made as usual. This is not the same as connecting nodes from two different **\psmatrix**'s. To do this, you must give the nodes names and refer to them by these names.

36 Complex examples

As this environment is very powerful and can be used for a lot of structured graphics (those of *grid*-like types), we give here two more complex examples.

First, a complex mathematical commutative diagram (rewritten from [17]):

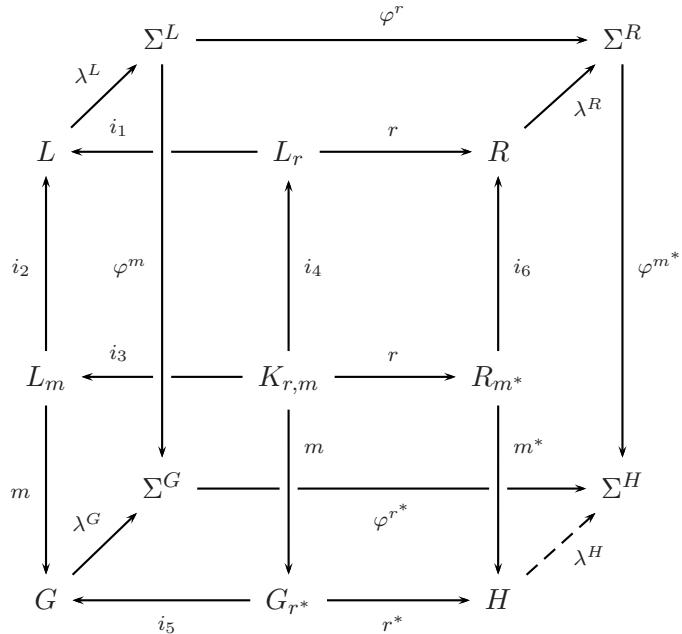
```

1 $ 
2 \psmatrix[rowsep=1,colsep=1]
3   % Nodes
4   % First line
5     & [name=SL] \Sigma^L &
6     & & [name=SR] \Sigma^R\! [0cm]
7   % Second line
8   [name=L] L & & [name=Lr] L_r
9     & & [name=R] R\! [1.5cm]
10  % Third line
11  [name=Lm] L_m & & [name=Krm] K_{r,m}
12    & & [name=Rm] R_{m^*}\! 
13    & & [name=SG] \Sigma^G &
14    & & & [name=SH] \Sigma^H\! [0cm]
15  % Fourth line
16  [name=G] G & & [name=Gr] G_{r^*}

```

```

17 & & [name=H] H
18 %
19 % Links
20 \everypsbox{\scriptstyle}
21 \psset{arrows=>,nodesep=2mm, border=3pt}%
22 \ncline{Lr}{R} ^{r}
23 \ncline{Krm}{Rm} ^{r}
24 \ncline{Gr}{H} _{r^*}
25 \ncline{Lr}{L} ^{[tpos=0.3]{i_1}}
26 \ncline{Krm}{Lm} ^{[tpos=0.3]{i_3}}
27 \ncline{Gr}{G} _{i_5}
28 \ncline{SL}{SR} ^{\varphi r}
29 \ncline{SG}{SH} _{\varphi^{r^*}}
30 \ncline{SR}{SH} >{\varphi^{r^* m^*}}
31 \ncline{SL}{SG} <{\varphi^{r^* m}}
32 \ncline{Lm}{G} <{m}
33 \ncline{Krm}{Lr} >{i_4}
34 \ncline{Rm}{H} >{[tpos=0.3]{m^*}}
35 \ncline{Lm}{L} <{i_2}
36 \ncline{Krm}{Gr} >{[tpos=0.3]{m}}
37 \ncline{Rm}{R} >{i_6}
38 \ncline{L}{SL} <{[tpos=0.3]{\lambda^L}}
39 \ncline{R}{SR} >{[tpos=0.6]{\lambda^R}}
40 \ncline{G}{SG} <{[tpos=0.3]{\lambda^G}}
41 \ncline[linestyle=dashed]{H}{SH} >{[tpos=0.6]{\lambda^H}}
42 \endpsmatrix
43 $
```



And second, a general connection diagram:

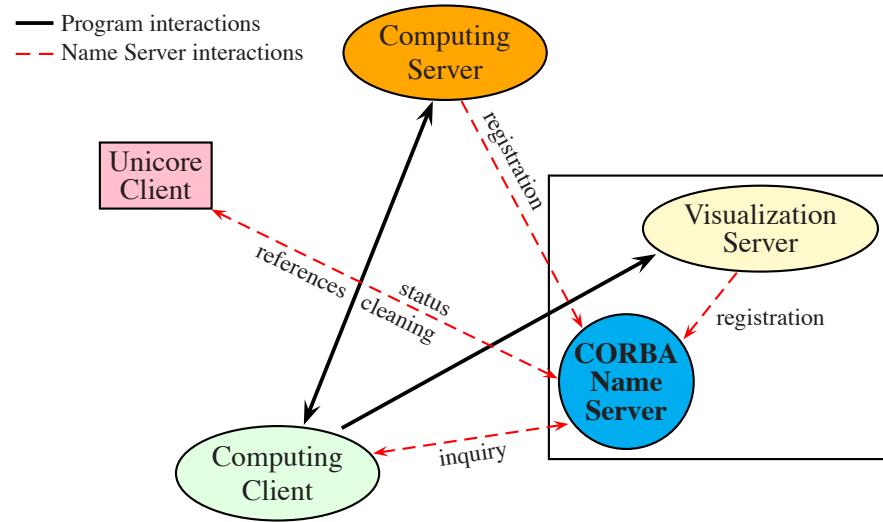
```

1 \newcommand{\PstComputingServer}{%
2 \psovalbox[fillstyle=solid,fillcolor=Orange]{%
```

```

3   \shortstack{Computing\\Server}}}
4
5 \newcommand{\PstUnicoreClient}{%
6 \psframebox[fillstyle=solid,fillcolor=Pink]{%
7   \shortstack{Unicore\\Client}}}
8
9 \newcommand{\PstVisualizationServer}{%
10 \psovalbox[fillstyle=solid,fillcolor=LemonChiffon]{%
11   \shortstack{Visualization\\Server}}}
12
13 \newcommand{\PstCorbaNameServer}{%
14 \pscirclebox[fillstyle=solid,fillcolor=cyan,framesep=0.02]{%
15   \bfseries\shortstack{CORBA\\Name\\Server}}}
16
17 \newcommand{\PstComputingClient}{%
18 \psovalbox[fillstyle=solid,fillcolor=PaleGreen]{%
19   \shortstack{Computing\\Client}}}
20
21 \rput[!]{(-0.5,6)}{%
22   \footnotesize
23   \shortstack[!]{%
24     \psline[linewidth=0.05,linecolor=black](-0.6,0.1)(-0.1,0.1)
25     Program interactions\\
26     \psline[linestyle=dashed,linecolor=red](-0.6,0.08)(-0.1,0.08)
27     Name Server interactions}}
28 %
29 \begin{psmatrix}[colsep=-0.5,rowsep=-0.5]
30   & & [name=ComputingServer] \PstComputingServer\\[1cm]
31   [name=Unicore] \PstUnicoreClient\\
32   & & \psframebox{%
33     \begin{psmatrix}[colsep=-0.7,rowsep=0]
34       & [name=Visualization]\PstVisualizationServer\\[5mm]
35       [name=CORBA] \PstCorbaNameServer
36     \end{psmatrix}}\\
37   & & [name=ComputingClient] \PstComputingClient
38 \end{psmatrix}
39 %
40 \ncline[arrowscale=1.5,linewidth=0.05]{<->}{%
41   ComputingServer}{ComputingClient}
42 \ncline[arrowscale=1.5,linewidth=0.05,offsetA=0.35,offsetB=0.5,
43   nodesepA=-0.35,nodesepB=0.2]{<->}{%
44   ComputingClient}{Visualization}
45 \psset{linestyle=dashed,linecolor=red,arrowscale=1.3,labelsep=0.06}%
46 \ncline[offsetA=0.08,nodesepA=0.07]{<->}{{Unicore}{CORBA}}
47   \naput[npos=0.6,nrot=:U]{\footnotesize status}
48   \nbput[npos=0.41,nrot=:U]{\footnotesize references \\ \\ cleaning}
49 \ncline[nodesepB=-0.2]{<->}{{ComputingServer}{CORBA}}
50   \naput[npos=0.3,nrot=:U]{\footnotesize registration}
51 \ncline[nodesepB=-0.15]{<->}{{Visualization}{CORBA}}
52   \naput{\footnotesize registration}
53 \ncline[offsetA=0.2,nodesepA=-0.05,nodesepB=-0.14]{<->}{%
54   ComputingClient}{CORBA}

```



VIII Trees

37 Overview



The node and node connections are perfect tools for making trees, but positioning the nodes using `\rput` would be rather tedious.²⁵ The file `pstree.tex` / `pstree.sty` contains a high-level interface for making trees.

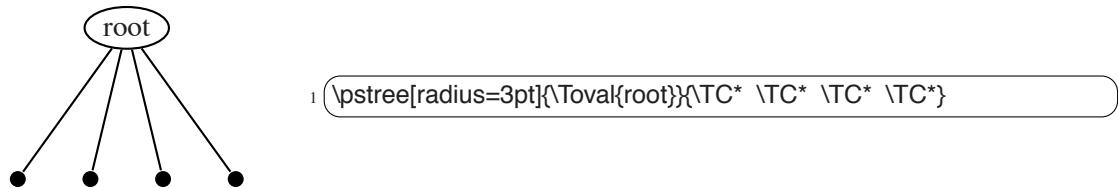
The tree commands are

```
\pstree{root}{successors}
\psTree{root} successors \endpsTree
```

These do the same thing, but just have different syntax. `\psTree` is the “long” version.²⁶ These macros make a box that encloses all the nodes, and whose baseline passes through the center of the root.

Most of the nodes described in Section 30 has a variant for use within a tree. These variants are called tree nodes, and are described in Section 38.

Trees and tree nodes are called *tree objects*. The *root* of a tree should be a single tree object, and the *successors* should be one or more tree objects. Here is an example with only nodes:

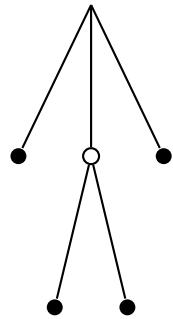


There is no difference between a terminal node and a root node, other than their position in the `\pstree` command.

Here is an example where a tree is included in the list of successors, and hence becomes subtree:

²⁵Unless you have a computer program that generates the coordinates.

²⁶L^AT_EX users can write `\begin{psTree}` and `\end{psTree}` instead.



```

1 \pstree[radius=3pt]{\Tp}{%
2   \TC*
3   \pstree{\TC}{\TC* \TC*}
4   \TC*}

```

38 Tree Nodes

For most nodes described in Section 30, there is a variant for use in trees, called a tree node. In each case, the name of the tree node is formed by omitting `node` from the end of the name and adding `T` at the beginning. For example, `\ovalnode` becomes `\oval`. Here is the list of such tree nodes:

```

\Tp*[par]
\Tc*[par]{dim}
\TC*[par]
\Tf*[par]
\Tdot*[par]
\Tr*[par]{stuff}
\TR*[par]{stuff}
\Tcircle*[par]{stuff}
\TCircle*[par]{stuff}
\Toval*[par]{stuff}
\Tdial*[par]{stuff}
\Tri*[par]{stuff}

```

The syntax of a tree node is the same as of its corresponding “normal” node, except that:

- There is always an optional argument for setting graphics parameters, even if the original node did not have one;
- There is no argument for specifying the name of the node;
- There is never a coordinate argument for positioning the node; and
- To set the reference point with `\Tr`, set the `ref` parameter.

Figure 1 gives a reminder of what the nodes look like.

The difference between `\Tr` and `\TR` (variants of `\rnode` and `\Rnode`, respectively) is important with trees. Usually, you want to use `\TR` with vertical trees because the baselines of the text in the nodes line up horizontally. For example:

```

1 \small
2 \psset{armB=1cm, levelsep=3cm, treesep=-3mm,
3     angleB=-90, angleA=90, nodesepA=3pt}
4 \def\s#1{\#1~{\tt\#1}}
5 \def\b#1{\#1{\tt\#1}}
6 \def\psedge#1#2{\ncangle{#2}{#1}}
7 \psTree[treenodesize=1cm]{\oval{Tree nodes}}
8 \s\Tp
9 \Tc{.5}~{\tt\string\Tc}
10 \s\TC
11 \psTree[levelsep=4cm,armB=2cm]{\Tp[edge=\ncline]}
12 \b\Tcircle
13 \s\Tdot
14 \TCircle[radius=1]{\tt\string\TCircle}
15 \Tn
16 \b\Toval
17 \b\Ttri
18 \b\Tdia
19 \endpsTree
20 \s\Tf
21 \b\Tr
22 \b\TR
23 \endpsTree

```

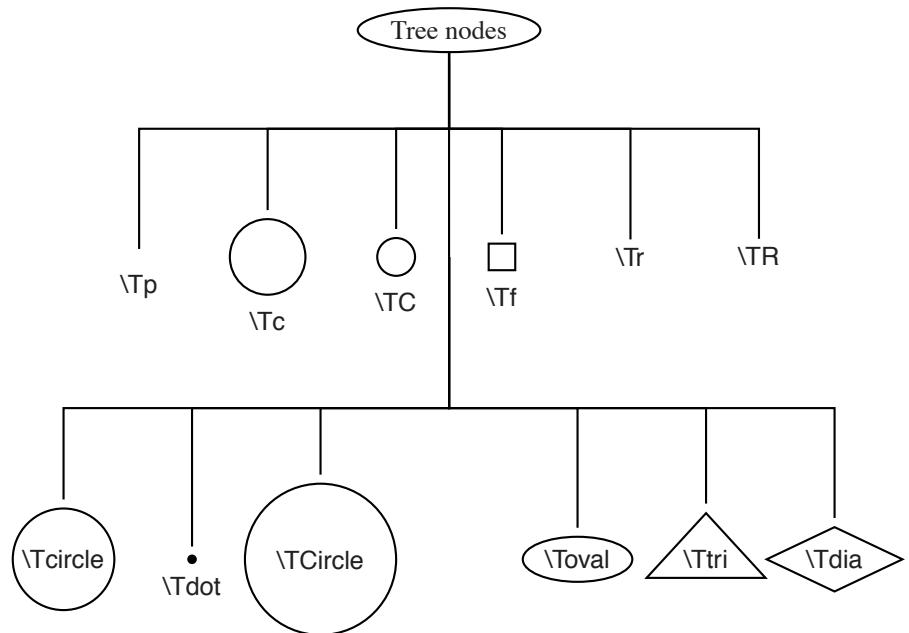
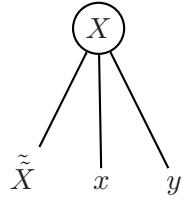


Figure 1: The tree nodes.

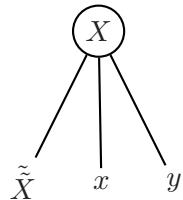


```

1 $ 
2 \pstree[nodesepB=3pt]{\Tcircle{X}}{%
3   \TR{\tilde{\tilde{X}}}
4   \TR{x}
5   \TR{y}}
6 $

```

Compare with this example, which uses \Tr:



```

1 $ 
2 \pstree[nodesepB=3pt]{\Tcircle{X}}{%
3   \Tr{\tilde{\tilde{X}}}
4   \Tr{x}
5   \Tr{y}}
6 $

```

There is also a null tree node:

\Tn

It is meant to be just a place holder. Look at the tree in Figure page ???. The bottom row has a node missing in the middle. \Tn was used for this missing node.

There is also a special tree node that doesn't have a “normal” version and that can't be used as the root node of a whole tree:

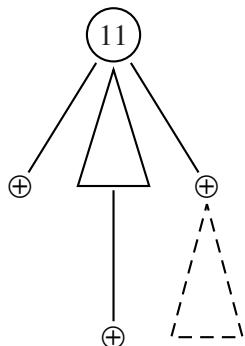
\Tfan*[par]

This draws a triangle whose base is

fansize=dim

Default: 1cm

and whose opposite corner is the predecessor node, adjusted by the value of **nodesepA** and **offsetA**. For example:



```

1 \pstree[dotstyle=oplus,dotsize=8pt,nodesep=2pt]{\Tcircle{11}}{%
2   \Tdot
3   \pstree{\Tfan}{\Tdot}
4   \pstree{\Tdot}{\Tfan[linestyle=dashed]}}
5

```

39 Tree orientation

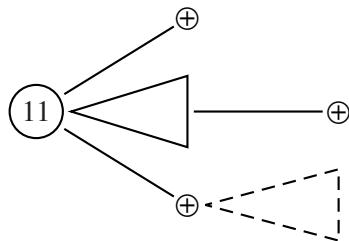
Trees can grow down, up, right or left, depending on the

treemode=D/U/R/L

Default: D

parameter.

Here is what the previous example looks like when it grows to the right:

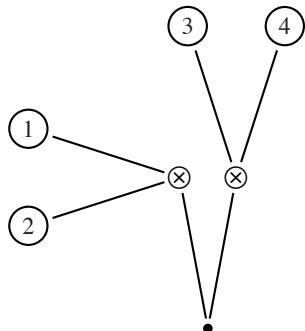


```

1 \pstree[dotstyle=oplus,dotsize=8pt,nodesep=2pt,treemode=R]
2 { \Tcircle{11}%
3   \Tdot
4   \pstree{\Tfan}{\Tdot}
5   \pstree{\Tdot}{\Tfan[linestyle=dashed]}}

```

You can change the **treemode** in the middle of the tree. For example, here is a tree that grows up, and that has a subtree which grows to the left:

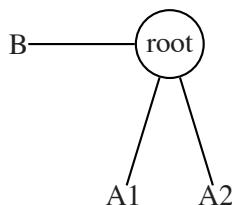


```

1 \footnotesize
2 \pstree[treemode=U,dotstyle=otimes,dotsize=8pt,nodesep=2pt]
3 { \Tdot%
4   \pstree[treemode=L]{\Tdot}{\Tcircle{1} \Tcircle{2}}
5   \pstree{\Tdot}{\Tcircle{3} \Tcircle{4}}}

```

Since you can change a tree's orientation, it can make sense to include a tree (*treeB*) as a root node (of *treeA*). This makes a single logical tree, whose root is the root of *treeB*, and that has successors going off in different directions, depending on whether they appear as a successor to *treeA* or to *treeB*.



```

1 \pstree{\pstree[treemode=L]{\Tcircle{root}}{\Tr{B}}}{%
2   \Tr{A1}
3   \Tr{A2}}

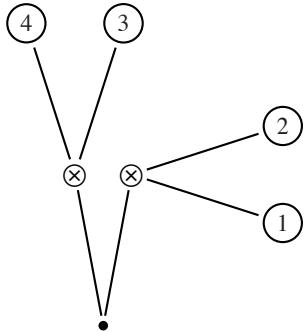
```

When the tree grows up or down, the successors are lined up from left to right in the order they appear in **\pstree**. When the tree grows to the left or right, the successors are lined up from top to bottom. As an afterthought, you might want to flip the order of the nodes. The

treeflip=true/false

Default: false

let's you do this. For example:



```
1 \footnotesize
2 \pstree[treemode=U,dotstyle=otimes,dotsize=8pt,
3   nodesep=2pt,treeflip=true]{\Tdot}{%
4   \pstree[treemode=R]{\Tdot}{\Tcircle{1} \Tcircle{2}}}
5   \pstree{\Tdot}{\Tcircle{3} \Tcircle{4}}}
```

Note that I still have to go back and change the **treemode** of the subtree that used to grow to the left.

40 The distance between successors

The distance between successors is

treesep=*dim*

Default: .75cm

The rest of this section describes ways to fine-tune the spacing between successors.

You can change the method for calculating the distance between subtrees by setting the

treefit=tight/loose

Default: tight

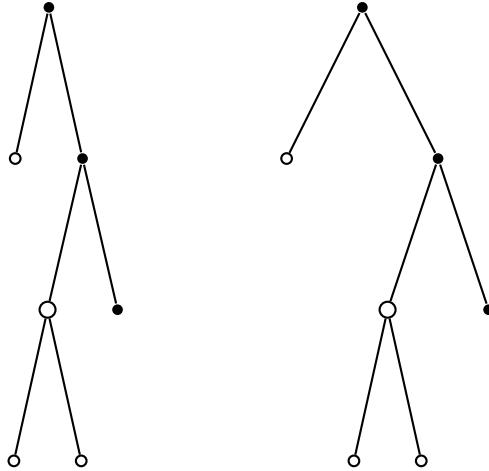
parameter. Here are the two methods:

tight When **treefit=tight**, which is the default, **treesep** is the minimum distance between each of the levels of the subtrees.

loose When **treefit=loose**, **treesep** is the distance between the subtrees' bounding boxes. Except when you have large intermediate nodes, the effect is that the horizontal distance (or vertical distance, for horizontal trees) between all the terminal nodes is the same (even when they are on different levels).²⁷

Compare:

²⁷When all the terminal nodes are on the same level, and the intermediate nodes are not wider than the base of their corresponding subtrees, then there is no difference between the two methods.



With **`treetype=loose`**, trees take up more space, but sometimes the structure of the tree is emphasized.

Sometimes you want the spacing between the centers of the nodes to be regular even though the nodes have different sizes. If you set

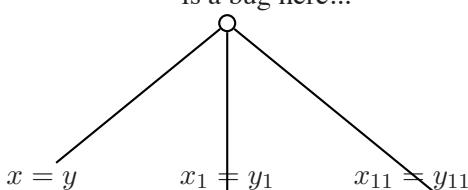
`treenodesize=dim`

Default: -1pt

to a non-negative value, then PSTricks sets the width (or height+depth for vertical trees) to **`treenodesize`**, *for the purpose of calculating the distance between successors*.

DG: It seems that there
is a bug here...

For example, ternary trees look nice when they are symmetric, as in the following example:

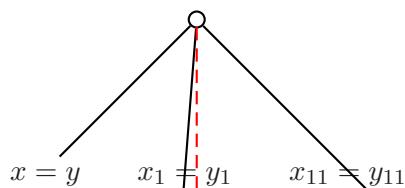


```

1 \pstree[nodesepB=-8pt,treenodesize=0.85]{\Tc{3pt}}%
2   \TR{$x=y$}
3   \TR{$x_1=y_1$}
4   \TR{$x_{11}=y_{11}$}

```

Compare with this example, where the spacing varies with the size of the nodes:



```

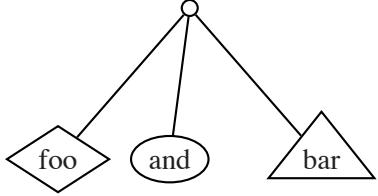
1 \pstree[nodesepB=-8pt]{\Tc{3pt}}%
2   \TR{$x=y$}
3   \TR{$x_1=y_1$}
4   \TR{$x_{11}=y_{11}$}

```

Finally, if all else fails, you can adjust the distance between two successors by inserting

`\tsep{dim}`

between them:



```

1 \pstree{\Tc{3pt}}{%
2   \Tdia{foo}
3   \tspac{-0.5}
4   \Toval{and}
5   \Ttri{bar}}
  
```

41 Spacing between the root and successors

The distance between the center lines of the tree levels is:

levelsep=**dim*

Default: 2cm

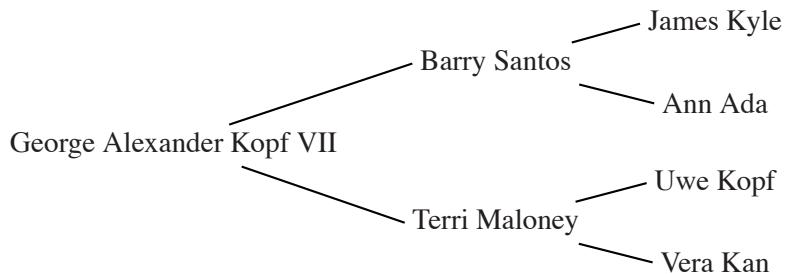
If you want the spacing between levels to vary with the size of the levels, use the * convention. Then **levelsep** is the distance between the bottom of one level and the top of the next level (or between the sides of the two levels, for horizontal trees).

Note: PSTricks has to write some information to your .aux file if using L^AT_EX, or to \jobname.pst otherwise, in order to calculate the spacing. You have to run your input file a few times before PSTricks gets the spacing right.

You are most likely to want to set **varlevelsep** to true in horizontal trees. Compare the following example:

```

1 \def\psedge#1#2{\ncdiag[nodesep=3pt,angleA=180,armA=0]{#2}{#1}}
2 \pstree[treemode=R,levelsep=*1cm]
3   {\TR{George Alexander Kopf VII}}%
4   \pstree[ref=c]{\Tr{Barry Santos}}{\Tr{James Kyle} \Tr{Ann Ada}}
5   \pstree[ref=c]{\Tr{Terri Maloney}}{\Tr{Uwe Kopf} \Tr{Vera Kan}}
  
```



with this one, were the spacing between levels is fixed:

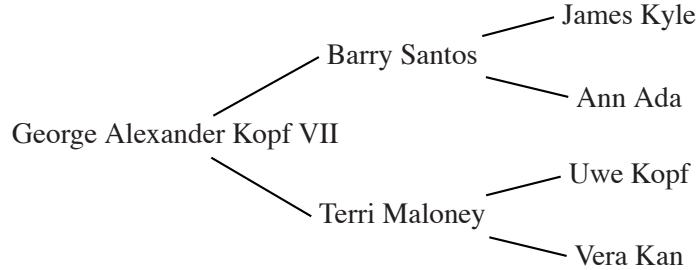
```

1 \def\psedge#1#2{\ncdiag[nodesep=3pt,angleA=180,armA=0]{#2}{#1}}
2 \pstree[treemode=R,levelsep=3cm]
3   {\Tr{George Alexander Kopf VII}}%
  
```

```

4 | \pstree[ref=c]{\Tr{Barry Santos}}{\Tr{James Kyle} \Tr{Ann Ada}}
5 | \pstree[ref=c]{\Tr{Terri Maloney}}{\Tr{Uwe Kopf} \Tr{Vera Kan}}

```



42 Edges

Right after you use a tree node command, `\pssucc` is equal to the name of the node, and `\pspred` is equal to the name of the node's predecessor. Therefore, you can draw a line between the node and its predecessor by inserting, for example,

```

1 | \ncline{\pspred}{\pssucc}

```

To save you the trouble of doing this for every node, each tree node executes

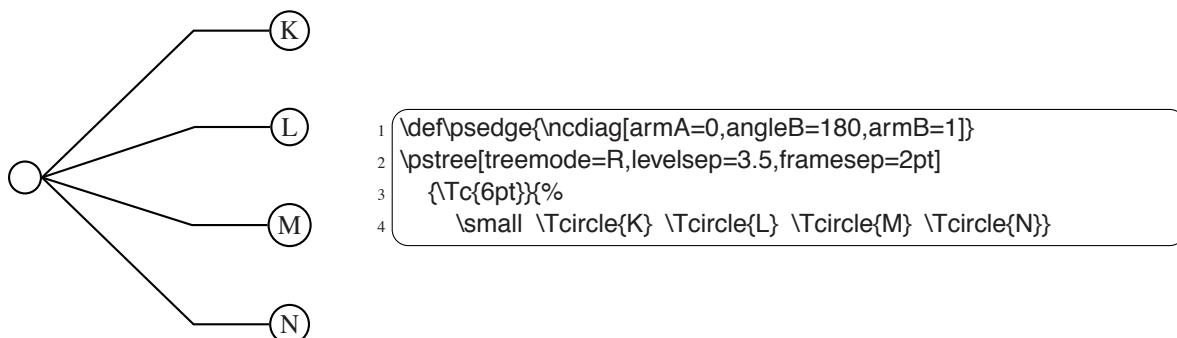
```

1 | \psedge{\pspred}{\pssucc}

```

The default definition of `\psedge` is `\ncline`, but you can redefine it as you please with `\def` or L^AT_EX's `\renewcommand`.

For example, here I use `\ncdiag`, with **armA=0**, to get all the node connections to emanate from the same point in the predecessor:²⁸



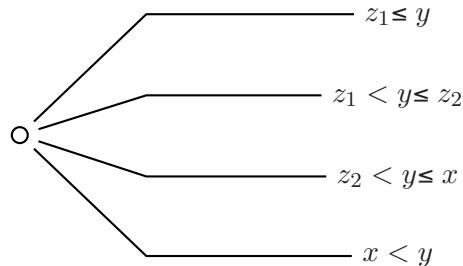
²⁸L^AT_EX users can instead type:

`\renewcommand{\psedge}{\ncdiag[armA=0,angleB=180,armB=1cm]}`

Here is an example with `\ncdiagg`. Note the use of a negative **armA** value so that the corners of the edges are vertically aligned, even though the nodes have different sizes:

```

1 $ 
2 \def\psedge#1#2{%
3   \ncdiagg[angleA=180,armA=-3,nodesep=4pt]{#2}{#1}%
4 % Or: \newcommand{\psedge}[2]{ ... }%
5 \pstree[treemode=R,levelsep=5]{\Tr{3pt}}{%
6   \Tr{z_1\leq y}%
7   \Tr{z_1<y\leq z_2}%
8   \Tr{z_2<y\leq x}%
9   \Tr{x<y}%
10 }$
```

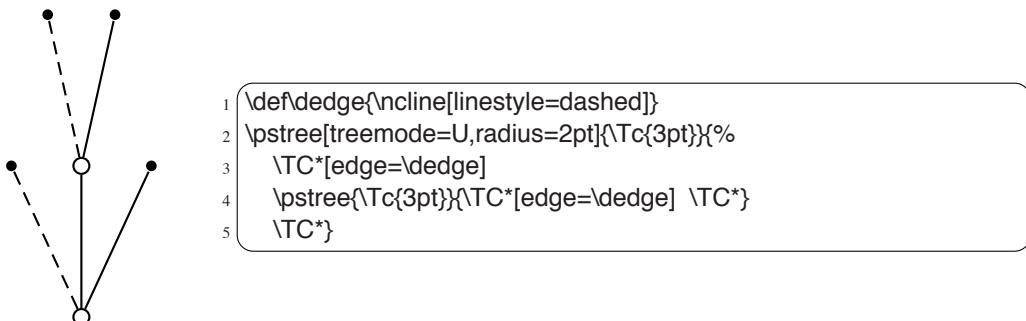


Another way to define `\psedge` is with the

edge=command

Default: \ncline

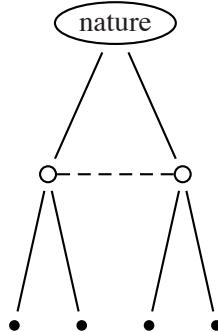
parameter. Be sure to enclose the value in braces {} if it contains commas or other parameter delimiters. This gets messy if your command is long, and you can't use arguments like in the preceding example, but for simple changes it is useful. For example, if I want to switch between a few node connections frequently, I might define a command for each node connection, and then use the **edge** parameter.



You can also set **edge=none** to suppress the node connection.

If you want to draw a node connection between two nodes that are not direct predecessor and successor, you have to give the nodes a name that you can

refer to, using the **name** parameter. For example, here I connect two nodes on the same level:

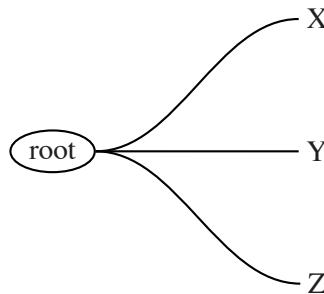


```

1 \pstree[nodesep=3pt,radius=2pt]{\Toval{nature}}{%
2   \pstree{\Tc[name=top]{3pt}}{\TC* \TC*}
3   \pstree{\Tc[name=bot]{3pt}}{\TC* \TC*}
4   \Incline[linestyle=dashed]{top}{bot}

```

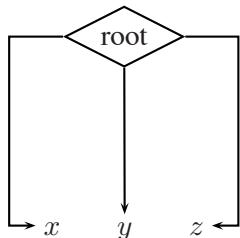
We conclude with the more examples.



```

1 \def\psedge{\nccurve[angleB=180,nodesepB=3pt]}
2 \pstree[treemode=R,treesep=1.5,levelsep=3.5]{\Toval{root}}{\Tr{X} \Tr{Y} \Tr{Z}}
3

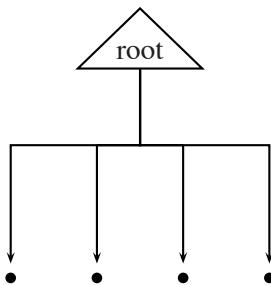
```



```

1 \pstree[arrows=->,nodesepB=3pt,xbbl=15pt,xbbr=15pt,
2   levelsep=2.5cm]{\Tdia{root}}{%
3   $%
4   \TR[edge=\ncbar{angle=180}]{x}
5   \TR{y}
6   \TR[edge=\ncbar]{z}
7 }

```



```

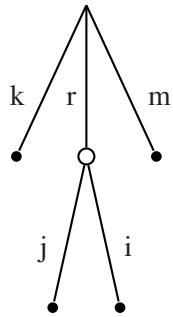
1 \psset{arrows=<,armB=1,levelsep=3,treesep=1,
2   angleB=-90,angleA=90,nodesepA=3pt}
3 \def\psedge{\ncangle{\#2}{\#1}}
4 \pstree[radius=2pt]{\Ttri{root}}{\TC* \TC* \TC* \TC*}

```

43 Edge and node labels

Right after a node, an edge has typically been drawn, and you can attach labels using **\ncput** **\tlput**, etc.

With `\tlput`, `\trput`, `\taput` and `\tbput`, you can align the labels vertically or horizontally, just like the nodes. This can look nice, at least if the slopes of the node connections are not too different.



```

1 \pstree[radius=2pt]{\Tp}{%
2   \psset{tpos=0.6}
3   \TC* \tlput{k}
4   \pstree{\Tc{3pt}} {\tlput[labelsep=3pt]{r},%
5     \TC* \tlput{j}}
6     \TC* \trput{i}}
7   \TC* \trput{m}}
  
```

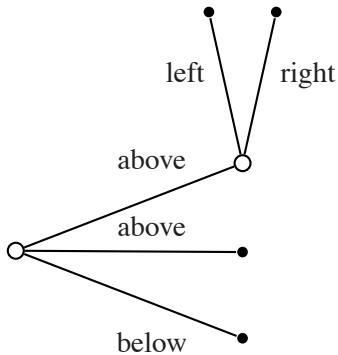
Within trees, the `tpos` parameter measures this distance from the predecessor to the successor, whatever the orientation of the tree. (Outside of trees it measures the distance from the top to bottom or left to right nodes.)

PSTricks also sets `shortput=tab` within trees. This is a special `shortput` option that should not be used outside of trees. It implements the following abbreviations, which depend of the orientation of the tree:

Short for:

<i>Char.</i>	<i>Vert.</i>	<i>Horiz.</i>
<code>^</code>	<code>\tlput</code>	<code>\taput</code>
<code>_</code>	<code>\trput</code>	<code>\tbput</code>

(The scheme is reversed if `treeflip=true`.)



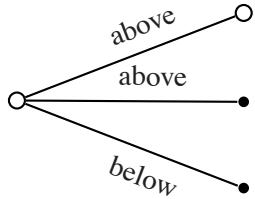
```

1 \psset{tpos=0.6}
2 \pstree[treemode=R,thistreesep=1, thislevelsep=3,
3   radius=2pt]{\Tc{3pt}}{%
4   \pstree[treemode=U,xbbr=20pt]{\Tc{3pt}}{\Tc{3pt}^{above},%
5     \TC*^{left},\TC*_{right},\TC*^{above},\TC*_{below}}}
6   \TC*^{above}
7   \TC*_{below}}
8
  
```

You can change the character abbreviations with

`\MakeShortTab{char1}{char2}`

The `\n*put` commands can also give good results:



```

1 \pssetnpos=0.6,nrot=:U
2 \pstree[treemode=R,thistreesep=1,thistlevelsep=3cm]{\Tc{3pt}}{%
3   \Tc{3pt}\nput{above}
4   \Tc*{2pt}\nput{above}
5   \Tc*{2pt}\nput{below}}

```

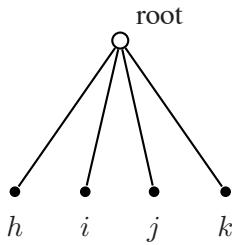
You can put labels on the nodes using `\nput`. However, `\pstree` won't take these labels into account when calculating the bounding boxes.

There is a special node label option for trees that does keep track of the bounding boxes:

`~*[par]{stuff}`

Call this a “tree node label”.

Put a tree node label right after the node to which it applies, before any node connection labels (but node connection labels, including the short forms, can follow a tree node label). The label is positioned directly below the node in vertical trees, and similarly in other trees. For example:



```

1 \pstree[radius=2pt]{\Tc{3pt}\nput{45}{\pssucc}{root}}{%
2   \TC*{$h$} \TC*{$i$} \TC*{$j$} \TC*{$k$}}

```

Note that there is no “long form” for this tree node label. However, you can change the single character used to delimit the label with

`\MakeShortTnput{char1}`

If you find it confusing to use a single character, you can also use a command sequence. E.g.,

```

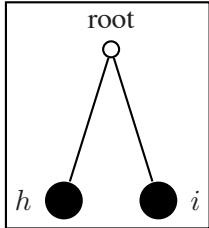
1 \MakeShortTnput{\tput}

```

You can have multiple labels, but each successive label is positioned relative to the bounding box that includes the previous labels. Thus, the order in which the labels are placed makes a difference, and not all combinations will produce satisfactory results.

You will probably find that the tree node label works well for terminal nodes, without your intervention. However, you can control the tree node labels by setting several parameters.

To position the label on any side of the node (left, right, above or below), set:

tnpos=/*r/a/b***Default:**

```

1 \psframebox{%
2   \pstree{\Tc{3pt}~[tnpos=a,tndepth=0,radius=4pt]{root}}{%
3     \TC*[tnpos=l]{$h$}
4     \TC*[tnpos=r]{$i$}}}

```

When you leave the argument empty, which is the default, PSTricks chooses the label position automatically.

To change the distance between the node and the label, set

tnsep=*dim***Default:**

When you leave the argument empty, which is the default, PSTricks uses the value of **labelsep**. When the value is negative, the distance is measured from the center of the node.

When labels are positioned below a node, the label is given a minimum height of

tnheight=*dim***Default:** `\ht\strutbox`

Thus, if you add labels to several nodes that are horizontally aligned, and if either these nodes have the same depth or **tnsep** is negative, and if the height of each of the labels is no more than **tnheight**, then the labels will also be aligned by their baselines. The default is `\ht\strutbox`, which in most T_EX formats is the height of a typical line of text in the current font. Note that the value of **tnheight** is not evaluated until it is used.

The positioning is similar for labels that go below a node. The label is given a minimum *depth* of

tndepth=*dim***Default:** `\dp\strutbox`

For labels positioned above or below, the horizontal reference point of the label, i.e., the point in the label directly above or below the center of the node, is set by the **href** parameter.

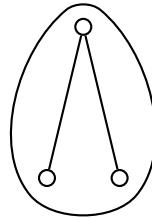
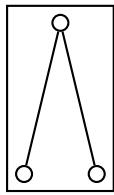
When labels are positioned on the left or right, the right or left edge of the label is positioned distance **tnsep** from the node. The vertical point that is aligned with the center of the node is set by

tnyref=*num***Default:**

When you leave this empty, **vref** is used instead. Recall that **vref** gives the vertical *distance* from the baseline. Otherwise, the **tnyref** parameter works like the **yref** parameter, giving the fraction of the distance from the bottom to the top of the label.

44 Framing

The **\pstreeframe** and **\pstreecurve** macros allow to frame trees or subtrees:

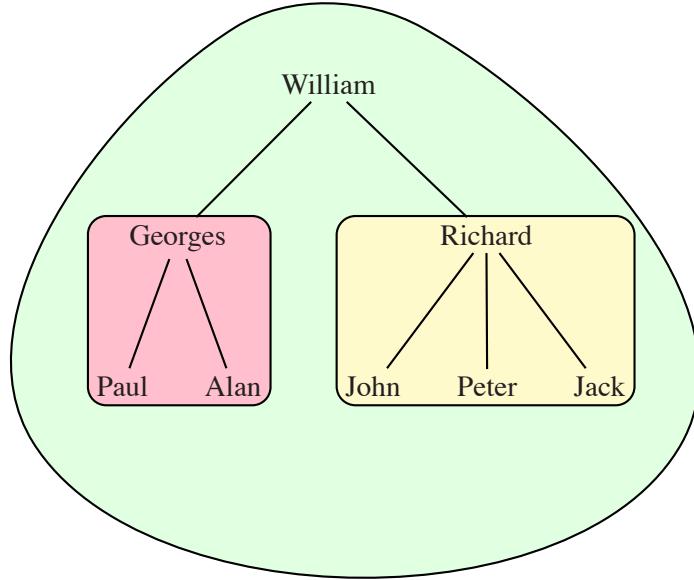


```

1 \pstreeframe{\pstree{\Tc{3pt}}{\Tc{3pt}\Tc{3pt}}}
2 \hspace{1cm}
3 \pstreecurve{\pstree{\Tc{3pt}}{\Tc{3pt}\Tc{3pt}}}
  
```

```

1 \pstreecurve[fillstyle=solid,fillcolor=PaleGreen,framesep=0.6]{%
2   \pstree[nodesep=3pt,framearc=0.2]{\TR{William}}{%
3     \pstreeframe[fillstyle=solid,fillcolor=Pink]{%
4       \pstree{\TR{Georges}}{\TR{Paul}\TR{Alan}}%
5     \pstreeframe[fillstyle=solid,fillcolor=LemonChiffon]{%
6       \pstree{\TR{Richard}}{\TR{John}\TR{Peter}\TR{Jack}}}}}}%
  
```



Note that the bounding boxes does not include the frame (at opposite that if we frame the tree with a macro like **\psframebox**), and that macros like **\psframebox** does not allow to frame subtrees.

45 Details

PSTricks does a pretty good job of positioning the nodes and creating a box whose size is close to the true bounding box of the tree. However, PSTricks does not take into account the node connections or labels when calculating the bounding boxes, except the tree node labels.

If, for this or other reasons, you want to fine tune the bounding box of the nodes, you can set the following parameters:

bb<i>l</i>=<i>dim</i>	Default:
bb<i>r</i>=<i>dim</i>	Default:
bb<i>h</i>=<i>dim</i>	Default:
bb<i>d</i>=<i>dim</i>	Default:
xbb<i>l</i>=<i>dim</i>	Default: 0
xbb<i>r</i>=<i>dim</i>	Default: 0
xbb<i>h</i>=<i>dim</i>	Default:
xbb<i>d</i>=<i>dim</i>	Default:

The `x` versions increase the bounding box by *dim*, and the others set the bounding box to *dim*. There is one parameter for each direction from the center of the node, `left`, `right`, `height`, and `depth`.

These parameters affect trees and nodes, and subtrees that switch directions, but not subtrees that go in the same direction as their parent tree (such subtrees have a profile rather than a bounding box, and should be adjusted by changing the bounding boxes of the constituent nodes).

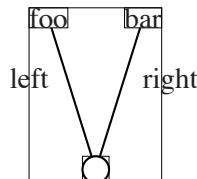
Save any fiddling with the bounding box until you are otherwise finished with the tree.

You can see the bounding boxes by setting the

showbbox=true/false	Default: false
----------------------------	-----------------------

parameter to `true`. To see the bounding boxes of all the nodes in a tree, you have to set this parameter before the tree.

In the following example, the labels stick out of the bounding box:

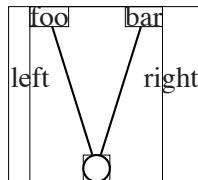


```

1 \psset{tpos=0.6,showbbox=true}
2 \pstree[treemode=U]{\Tc{5pt}}{%
3   \TR{foo}^{\left}
4   \TR{bar}_{\right}}

```

Here is how we fix it:

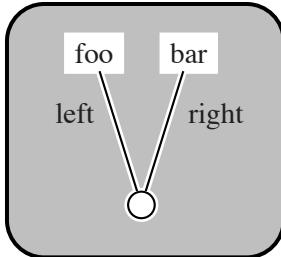


```

1 \psset{tpos=0.6,showbbox=true}
2 \pstree[treemode=U,xbbl=8pt,xbbr=14pt]{\Tc{5pt}}{%
3   \TR{foo}^{\left}
4   \TR{bar}_{\right}}

```

Now we can frame the tree:



```

1 \psframebox[fillstyle=solid,fillcolor=lightgray,framesep=14pt,
2   linearc=14pt,cornersize=absolute,linewidth=1.5pt]{%
3   \psset{tpos=0.6, border=1pt, nodesepB=3pt}
4   \pstree[treemode=U,xbbl=8pt,xbbr=14pt]{%
5     \Tc[fillcolor=white,fillstyle=solid]{5pt}}{%
6     \TR*{foo}^{\left\{left\right\}}
7     \TR*{bar}_{\left\{right\right\}}}}
```

We would have gotten the same result by changing the bounding box of the two terminal nodes.

To skip levels, use

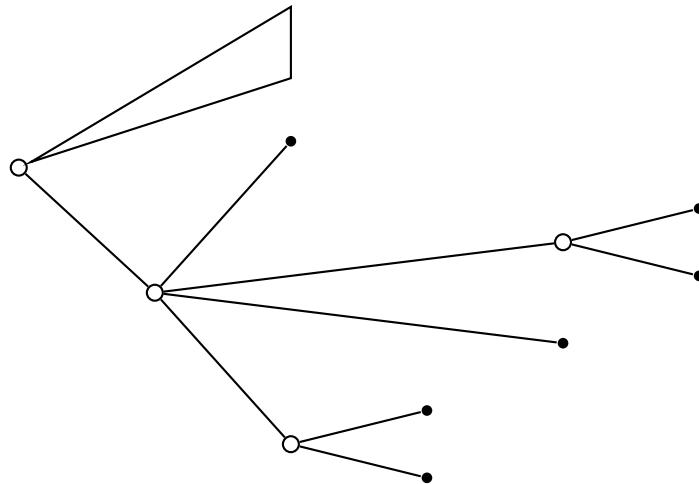
```

\skiplevel*[par]{nodes or subtrees}
\skiplevels*[par]{int} nodes or subtrees \endskiplevels
```

These are kind of like subtrees, but with no root node.

```

1 \pstree[treemode=R,levelsep=1.8,radius=2pt]{\Tc{3pt}}{%
2   \skiplevel{\Tfan}
3   \pstree{\Tc{3pt}}{%
4     \TC*
5     \skiplevels{2}
6     \pstree{\Tc{3pt}}{\TC* \TC*}
7     \TC*
8   \endskiplevels
9   \pstree{\Tc{3pt}}{\TC* \TC*}}}
```



The profile at the missing levels is the same as at the first non-missing level. You can adjust this with the bounding box parameters. You get greatest control if you use nested **\skiplevel** commands instead of **\skiplevels**.

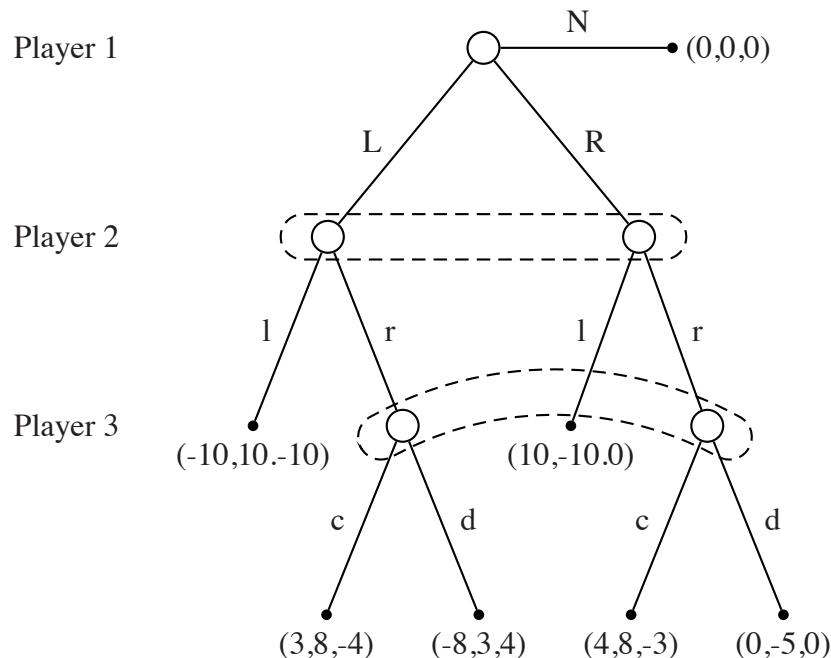
```

1 \large
2 \psset{radius=6pt,dotsize=4pt}
```

```

3 | \pstree{thislevelsep=0,edge=none,levelsep=2.5cm}{\Tn}{%
4 |   \pstree{\TR{Player 1}}{\pstree{\TR{Player 2}}{\TR{Player 3}}}
5 |   \psset{edge=\ncline}
6 |   \pstree
7 |     {\pstree{treemode=R}\TC}{\Tdot \sim{(0,0,0)} \N}%
8 |     \pstree{\TC[name=A]}{\L}%
9 |       \Tdot \sim{(-10,10,-10)} \I
10 |      \pstree{\TC[name=C]}{\R}%
11 |        \Tdot \sim{(3,8,-4)} \C
12 |        \Tdot \sim{(-8,3,4)} \D
13 |      \pstree{\TC[name=B]}{\R}%
14 |        \Tdot \sim{(10,-10,0)} \I
15 |        \pstree{\TC[name=D]}{\R}%
16 |          \Tdot \sim{(4,8,-3)} \C
17 |          \Tdot \sim{(0,-5,0)} \D}%
18 | \ncbox[linearc=0.3,boxsize=0.3,linestyle=dashed,nodesep=0.4]{A}{B}
19 | \ncarcbox[linearc=0.3,boxsize=0.3,linestyle=dashed,arcangle=25,
20 |   nodesep=0.4]{D}{C}

```



46 The scope of parameter changes

edge is the only parameter which, when set in a tree node's parameter argument, affects the drawing of the node connection (e.g., if you want to change the **nodesep**, your edge has to include the parameter change, or you have to set it before the node).

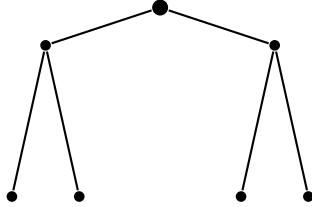
As noted at the beginning of this section, parameter changes made with **\pstree** affect all subtrees. However, there are variants of some of these

parameters for making local changes, i.e, changes that affects only the current level:

thistreesep=dim
thisreenodesize=dim
thistreefit=tight/loose
thislevelsep=dim

Default:
Default:
Default:
Default:

For example:



```

1 \pstree[thislevelsep=0.5,thistreesep=2,radius=2pt]{\TC*{3pt}}{
2   \pstree{\TC*{\TC* \TC*}}{
3     \pstree{\TC*{\TC* \TC*}}{ }
  }

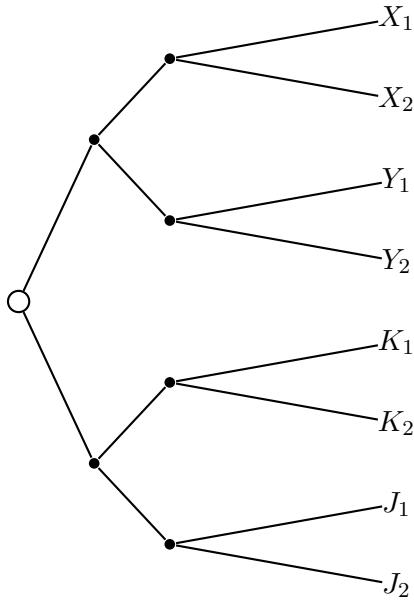
```

There are some things you may want set uniformly across a level in the tree, such as the **levelsep**. At level n , the command `\pstreehook<roman(n)>` (e.g., `\pstreehookii`) is executed, if it is defined (the root node of the whole tree is level 0, the successor tree objects and the node connections from the root node to these successors is level 1, etc.). In the following example, the **levelsep** is changed for level 2, without having to set the **thislevelsep** parameter for each of the three subtrees that make of level 2:

```

1 \[
2 \def\pstreehookii{\psset{thislevelsep=3cm}}
3 \pstree[treemode=R,levelsep=1cm,radius=2pt]{\TC{4pt}}{
4   \pstree{\TC* \% }{
5     \pstree{\TC*{\Tr{X_1} \Tr{X_2}}}{ }
6     \pstree{\TC*{\Tr{Y_1} \Tr{Y_2}}}{ }
7     \pstree{\TC* \% }{
8       \pstree{\TC*{\Tr{K_1} \Tr{K_2}}}{ }
9       \pstree{\TC*{\Tr{J_1} \Tr{J_2}}}{ }
10      }
  }
\]

```



47 Complex examples

As trees are a very often representation structure used in many areas, we give here three more complex examples.

First a general organization tree:

```

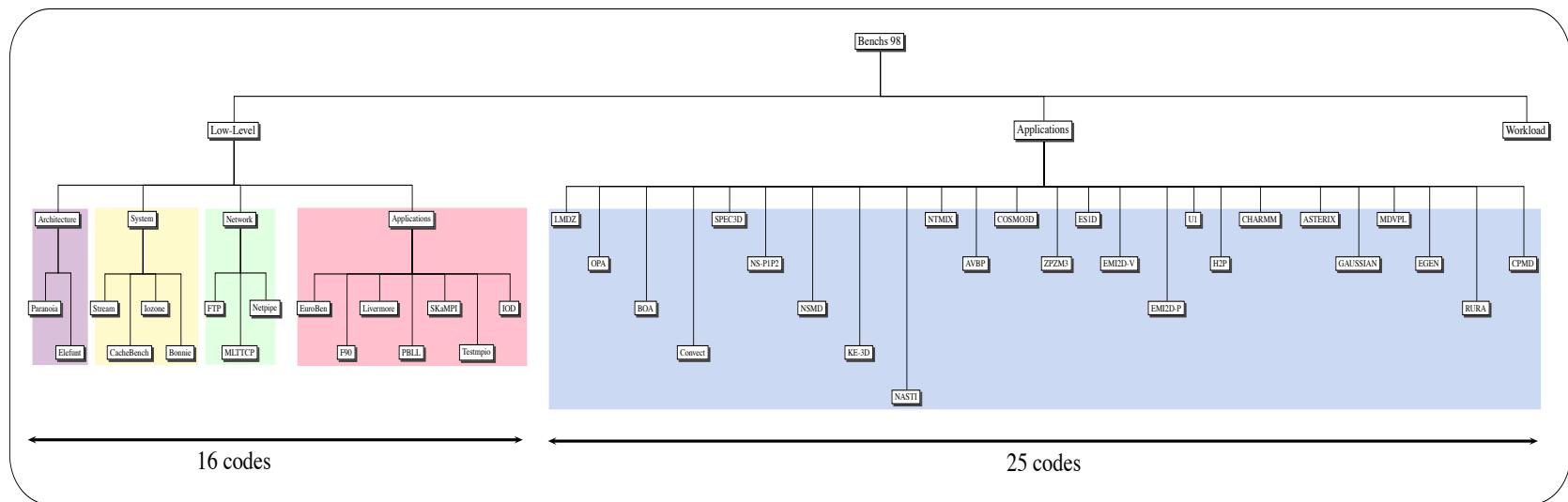
1 % Connection style
2 \def\psedge{\ncangle[angleA=90,angleB=-90]{#2}{#1}}
3
4 \def\DirBa{\Tr{\psshadowbox{\footnotesize #1}}}
5 \def\DirBx{%
6   \pstree[thislevelsep=2,treenodesize=0.6]{\Tn}{\DirBa{#1}}}
7 \def\DirBb{\DirBx{1.5}}
8 \def\DirBc{\DirBx{3}}
9 \def\DirBd{\DirBx{4.5}}
10 \def\DirBe{\DirBx{6}}
11
12 \psset{levelsep=3,armB=1.5}%
13
14 \rotateleft{%
15   \scalebox{2.7}{%
16     \psframebox[framesep=0.8,framearc=0.2]{%
17       \pstree[xbbd=2.5]{\DirBa{\large Benchs 98}}{%
18         \pstree[treenodesize=0.3]{\DirBa{\large Low-Level}}{%
19           \input[labelsep=10.5]{-90}{\pssucc}{%
20             \psline[linewidth=0.1]{<->}{-7,1)(13.5,1}%
21             \Huge 16 codes}}%
22       \pstreeframe*[linecolor=Thistle]{%
23     }
24   }
25 }

```

```

24          \pstree{\DirBa{Architecture}}
25              {\DirBa{Paranoia}
26                  \DirBb{Elefunt}}}
27          \pstreeframe*[linecolor=LemonChiffon]{%
28              \pstree{\DirBa{System}}
29                  {\DirBa{Stream}
30                      \DirBb{CacheBench}
31                      \DirBa{lozone}
32                      \DirBb{Bonnie}}}
33          \pstreeframe*[linecolor=PaleGreen]{%
34              \pstree{\DirBa{Network}}
35                  {\DirBa{FTP}
36                      \DirBb{MLTCP}
37                      \DirBa{Netpipe}}}
38          \pstreeframe*[linecolor=Pink]{%
39              \pstree[treenodesize=0.6]{\DirBa{Applications}}
40                  {\DirBa{EuroBen}
41                      \DirBb{F90}
42                      \DirBa{Livermore}
43                      \DirBb{PBLL}
44                      \DirBa{SKaMPI}
45                      \DirBb{Testmpio}
46                      \DirBa{IOD}}}}
47          \pstree{\DirBa{\large Applications}}%
48              \nput[labelsep=10.5]{-90}{\pssucc}{%
49                  \psline[linewidth=0.1]{<->}{(-19,1)(22,1)
50                      Huge 25 codes}}
51          \pstreeframe*[linecolor=LightBlue]{%
52              \pstree[thislevelsep=0]{\Tn}
53                  {\DirBa{LMDZ}
54                      \DirBb{OPA}
55                      \DirBc{BOA}
56                      \DirBd{Convect}
57                      \DirBa{SPEC3D}
58                      \DirBb{NS-P1P2}
59                      \% ...
60                      \DirBb{CPMD}}}}
61          \tspacet{17}
62          \DirBa{\large Workload}}}}}

```



And, second, an example of a complex genealogical tree, coming from Fran oise Coustillas and Louis Rigot.

```

1 % Default edge
2 \renewcommand{\psedge}{\ncangle[angleA=-90]}
3
4 % Macro to define one person
5 \newcommand{\Person}[3]{\%
6 \TR[#1]\%
7 \setlength{\tabcolsep}{0mm}%
8 \begin{tabular}{t#2#3}\end{tabular}\}}
9
10 % Definition of people
11 \newcommand{\CatherineVerrier}\%
12 \Person{c}\%
13   \rnode{NodeTempA}\%
14     Catherine \textsc{errier} (\oldstylenums{1741}, Feyzin –
15     \oldstylenums{1826}, Lyon)\hspace{1mm}\\
16     fille de Marin et de Beno eete Langlois, domestique\ \
17   \rnode{NodeTempB}\%
18     \hspace{1mm}x \oldstylenums{1764}, V nissieux\ \
19     Jean-Baptiste \textsc{uffard}\ \
20     (v. \oldstylenums{1739}, Ardon,
21     auj. Ch e2illon-en-Michaille ? –\ \
22     \oldstylenums{1777}, La Guilloti re, auj. Lyon)\ \
23     domestique, postillon, m\textsuperscript{t}extsuperscript{d}  picier, revendeur\}
24
25 \newcommand{\MarieVerrier}\%
26 \Person[edge=\ncangle[angleA=0]]{c}\%
27   Marie \textsc{errier}\ \
28   % ...
29
30 \newcommand{\MadeleineBuffard}\%
31 \Person[edge=\ncangles[angleA=180,armA=2.2cm]]{c}\%
32   Magdelaine \textsc{uffard}\ \
33   % ...
34 \scriptsize
35
36 {\Large Parent  avec Paul Vigi re d'Anval}
37 {\Large des parties signataires de son acte de bapt ame}
38
39 \fbox{\makebox[\textwidth]\%
40   \pstree[levelsep=1.5,baselinekip,treesep=-0.1,
41     armB=0.6\baselinekip,angleB=90,linewidth=0.015]
42     {\CatherineVerrier
43       \gdef\pssucc{NodeTempB}\}
44     \pstree{\MadeleineBuffard}
45       {\ClaudeMorenne}
46     \pstree[thistreesep=1mm]
47       {\AntoinetteBuffard
48         \gdef\pssucc{NodeTempC}\}
49       {\AndreCreuzet

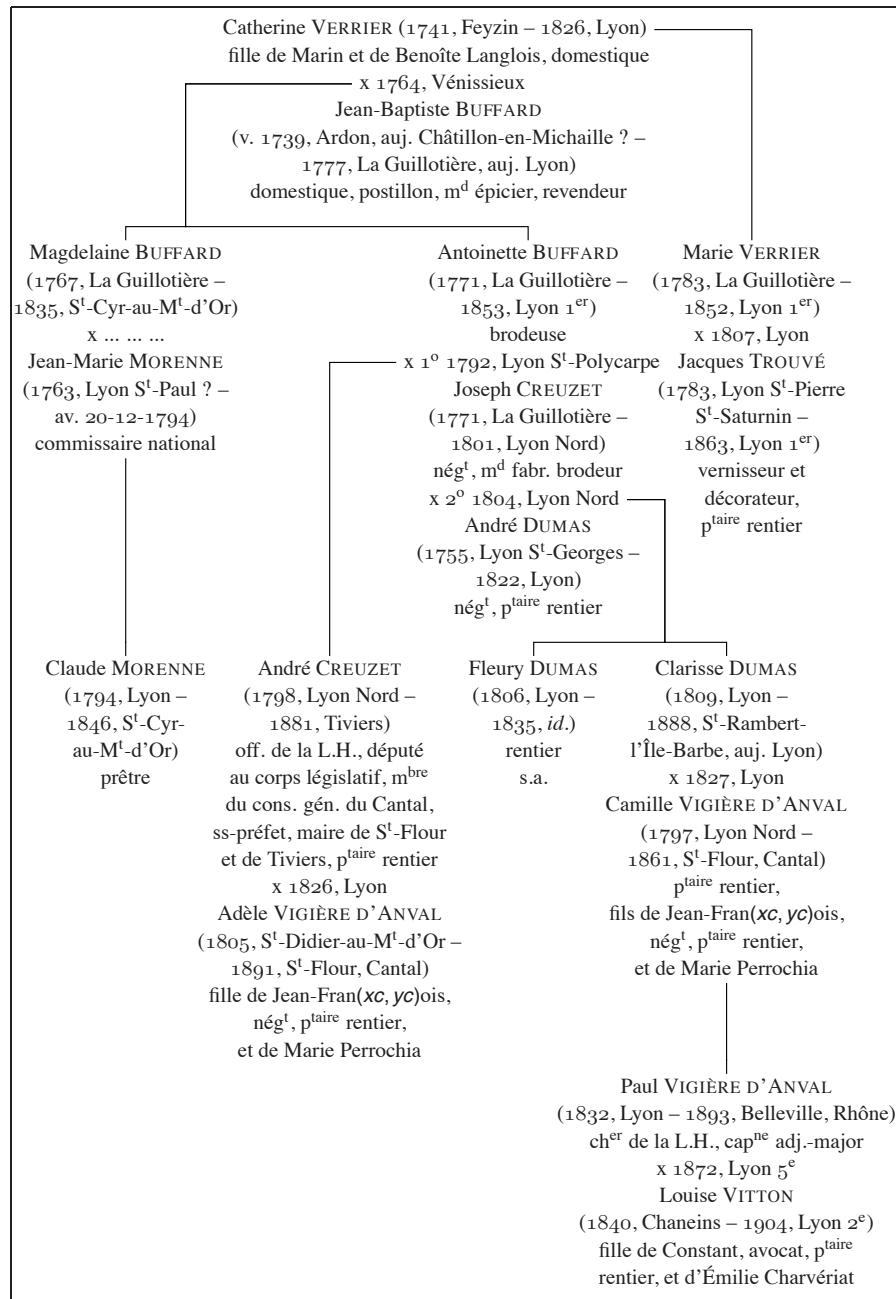
```

```

50 \gdef\pspred{NodeTempD}%
51 \FleuryDumas
52 \pstree[levelsep=4\baselineskip]
53 {\ClarisseDumas}
54 {\PaulVigiere}}
55 \gdef\pspred{NodeTempA}%
56 \MarieVerrier}}}

```

Parenté avec Paul Vigière d'Anval des parties signataires de son acte de baptême



```

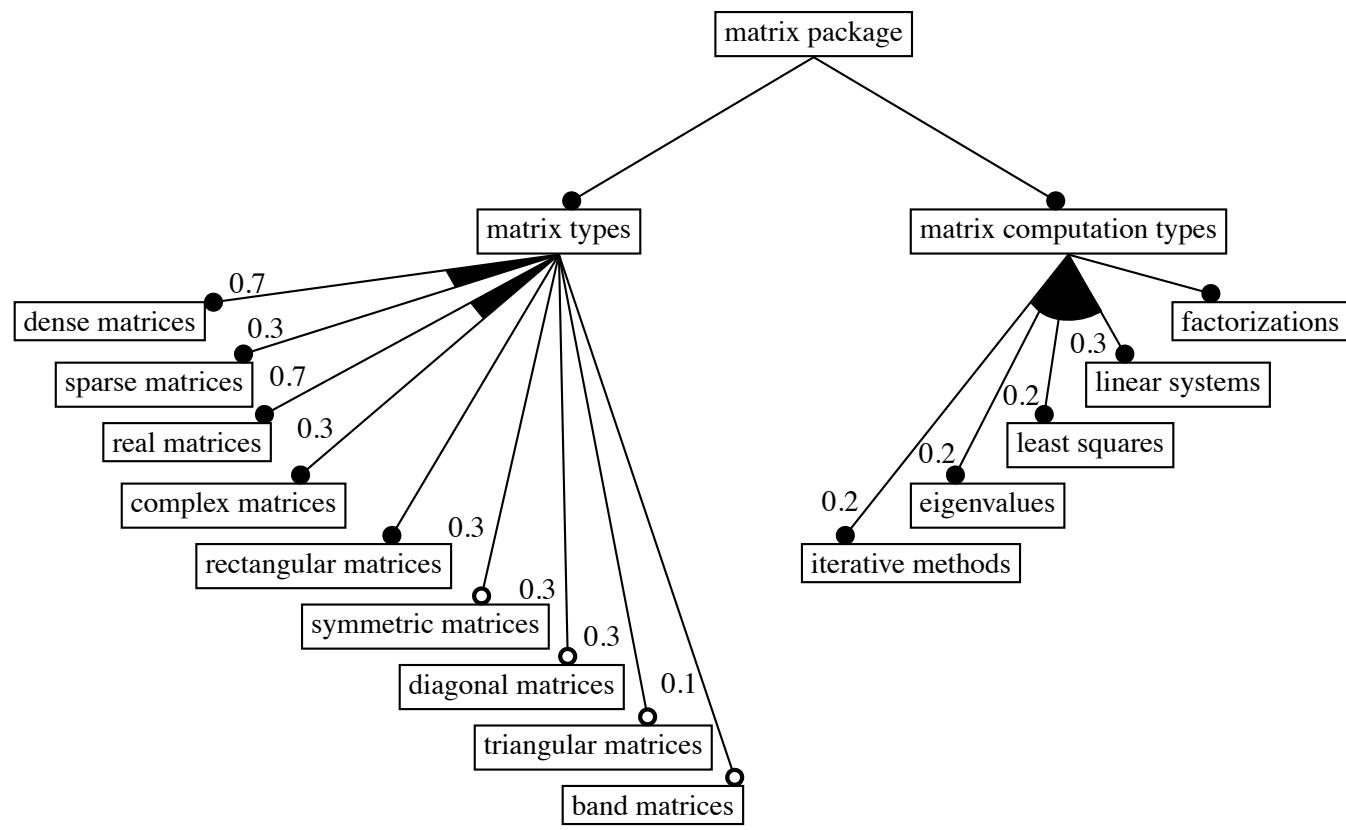
1 \def\MyNodeA{\@ifnextchar[{]{\MyNodeA@i}{\MyNodeA@i[]}}
2 \def\MyNodeA@i[#1]{\Tr[#1]{\psframebox{#2}}}
3 \def\MyNodeB{\@ifnextchar[{]{\MyNodeB@i}{\MyNodeB@i[]}}
4 \def\MyNodeB@i[#1]{\TR[#1]{\psframebox{#2}}}
5 \def\MyNodeC{\@ifnextchar[{]{\MyNodeC@i}{\MyNodeC@i[]}}
6 \def\MyNodeC@i[#1]{\Tr[#1]{\rnode[b]{#2}{\psframebox{#3}}}}
7
8 \SpecialCoor
9
10 \pstree[arrows={-*},arrowscale=2,nodesepB=0.1]
11   {\MyNodeA[ref=b]{matrix package}}
12   \pstree[treesep=-2,levelsep=1.5]
13     {\MyNodeC[ref=t]{Types}{matrix types}}
14     \psset{levelsep=0.8,labelsep=0.1}%
15     \def\pspred{Types}%
16     \MyNodeB[href=-0.4]%
17       \Rnode[href=-0.4]{Dense}{dense matrices}}
18       \tput[tpos=0.44]{0.7}
19     \skipto{1}
20       \MyNodeB\Rnode{Sparse}{sparse matrices}}
21       \tput[tpos=0.58]{0.3}
22     \endskipto
23     \pspolygon*[nodesep=3.3]{\pspred}{Dense}(\pspred)
24       ([nodesep=2.8]{\pspred}{Sparse})
25     \skipto{2}
26       \MyNodeB[href=0.3]%
27         \Rnode[href=0.3]{Real}{real matrices}}
28         \tput[tpos=0.65]{0.7}
29     \endskipto
30     \skipto{3}
31       \MyNodeB[href=0.3]%
32         \Rnode[href=0.3]{Complex}{complex matrices}}
33         \tput[tpos=0.7]{0.3}
34     \endskipto
35     \pspolygon*[nodesep=2.75]{\pspred}{Real}(\pspred)
36       ([nodesep=2.3]{\pspred}{Complex})
37     \skipto{4}
38       \MyNodeB[href=0.4]{rectangular matrices}
39     \endskipto
40     \psset{arrows={-o},tpos=0.78}%
41     \skipto{5}
42       \MyNodeB[href=0.4]{symmetric matrices}
43         \tput[tpos=0.74]{0.3}
44     \endskipto
45     \skipto{6}
46       \MyNodeB[href=0.5]{diagonal matrices}
47         \tput{0.3}
48     \endskipto
49     \skipto{7}
50       \MyNodeB[href=0.5]{triangular matrices}
51         \tput{0.3}
52     \endskipto

```

```

53      \skiplevels{8}
54      \MyNodeB[href=1]{band matrices}
55      \tput[labelsep=0]{0.1}
56      \endskiplevels
57      \tspc{6}\Tn
58      \pstree[treesep=-3.5,levelsep=1.5]
59      {\MyNodeC[ref=t]{Computation}%
60       matrix computation types}}
61      {\def\pspred{Computation}%
62       \MyNodeB[href=0.6]{factorizations}
63       \psset{levelsep=0.8,labelsep=0.05,tpos=0.8}%
64       \skiplevels{1}
65       \MyNodeB[href=-0.4]{%
66        \Rnode[href=-0.4]{Linear}{linear systems}}
67       \tput[tpos=0.7]{0.3}
68       \endskiplevels
69       \skiplevels{2}
70       \MyNodeB[href=-0.6]{least squares}
71       \tput[tpos=0.75]{0.2}
72       \endskiplevels
73       \skiplevels{3}
74       \MyNodeB[href=-0.6]{eigenvalues}
75       \tput{0.2}
76       \endskiplevels
77       \skiplevels{4}
78       \MyNodeB[href=-0.8]{%
79        \Rnode[href=-0.8]{Iterative}{iterative methods}}
80       \tput{0.2}
81       \endskiplevels
82       \pnode[nodesep=0.8]{\pspred}{Linear}{LinearB}
83       \pnode[nodesep=3.3]{\pspred}{Iterative}{IterativeB}
84       \pscustom*[arrows=]{%
85        \psline(IterativeB)(\pspred)(LinearB)
86        \ncarc[arcangle=40]{LinearB}{IterativeB}}}}

```



IX

Filling and Tiling²⁹

48 Overview



The file `pst-fill.tex/pst-fill.sty` contains a high-level interface for simple drawing of various kinds of filling and area tiling.

We use the word *filling* to describe the operation which consists of filling a defined area by a pattern (or a composition of patterns), and *tiling* as the operation which is like filling, but with control of the starting point (we use the upper left corner), where the pattern is positioned relative to this point. There is an essential difference between the two modes, as without control of the starting point we cannot create the *tilings* (sometimes called *tesselations*) used in many fields of Art and Science.

Tilings are a wide and difficult field of mathematics,³⁰ and this package is limited to simple ones, mainly *monohedral* tilings with one prototile (which nevertheless can be composite). With some experience and wiliness we can do more, and easily obtain quite sophisticated results, but obviously hyperbolic tilings like the famous Escher ones or aperiodic tilings like the Penrose ones are not within the capabilities of this package. For more complex needs, we must use low level and more painfull techniques, with the basic `\multido` and `\multirput` macros.

This package defines two modes are defined, called respectively *manual* and *automatic*. For both, the pattern is generated on contiguous positions in a large area which includes the region to fill, which is later cut to the required dimensions by a clipping mechanism. In the first mode, the pattern is explicitely inserted in the PostScript output file each time. In the second, the result is the same but with a single insertion of the pattern and a repetition done by PostScript. Control over the starting point was lost, so it allowed only *filling* a region and not to *tiling* it.

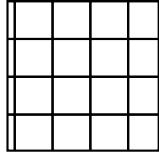
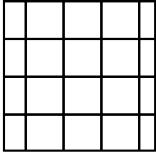
Using the following utility macro:

```
1 \def\Square{%
2 \pspicture(0.5,0.5)\psframe[dimen=middle](0.5,0.5)\endpspicture}
```

²⁹This chapter is an adaptation of the documentation for this package written in 1997 by Denis Girou.

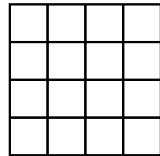
³⁰For an extensive presentation of tilings, in their history and usage in many fields, see the reference book by Branko Grünbaum and Geoffrey Shephard, *Tilings and Patterns* [28].

the difference between the two modes is shown here; *filling*:



```
1 \psboxfill{\Square}
2
3 \begin{pspicture}(2,2)
4   \psframe[fillstyle=boxfill](2,2)
5 \end{pspicture}
6 \hspace{5mm}
7 \begin{pspicture}(2,2)
8   \psframe[fillstyle=boxfill](2,2)
9 \end{pspicture}
```

where, as you can see, the initial position is arbitrary and depends on the current point, and *tiling*:



```
1 \psboxfill{\Square}
2 \psframe[fillstyle=boxfill](2,2)
```

It is clear that filling is very restrictive compared to tiling, as the desired effect very often requires the possibility of controlling the starting point. To load the extended mode, L^AT_EX users must load the package with the *tiling* parameter and plain T_EX and Con^TE_Xt users must define the following macro after loading the package: `\def\PstTiling{true}`

There is little reason to use the *manual* mode, apart very special cases where the *automatic* one cannot work, because it has the very big disadvantage of requiring very large resources, in disk space and subsequently in printing time (a small tiling can sometimes require several megabytes in *manual* mode!) Only one case is known, when some kinds of EPS files are used, such as the ones produced by partial screen dumps.

Note that in all of our next examples, we will always use *tiling* mode, and also that tilings are drawn from left to right and top to bottom, which can have be important in some circumstances.

And PostScript programmers may be interested to know that, even in *automatic* mode, the iterations of the pattern are managed directly by the PostScript code of the package, which uses only PostScript Level 1 operators. The special ones introduced in Level 2 for drawing patterns are not used.

49 Usage

To do a tiling, we just have to define the pattern with the `\psboxfill` macro and to use the **fillstyle** boxfill.

First, for convenience, we define a simple \Tiling macro, which will simplify our next examples:

```

1 \def\Tiling#1#2{%
2 \edef\Temp{\#1}%
3 \pspicture#2
4 \ifx\Temp\empty
5   \psframe[fillstyle=boxfill]#2
6 \else
7   \psframe[fillstyle=boxfill,\#1]#2
8 \fi
9 \endpspicture}

```

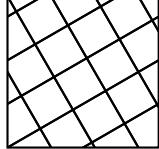
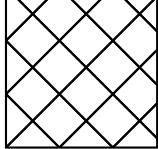
There are several parameters available to change the way the filling/tiling is defined, and one debugging option.

fillangle=angle

Default: 0

The rotation applied to the patterns.

In this case, we must force the tiling area to be noticeably larger than the area to cover, to be sure that the defined area will be covered after rotation.



```

1 \psboxfill{\Square}
2
3 \Tiling{fillangle=45}{(2,2)}
4 \hspace{5mm}
5 \Tiling{fillangle=-60}{(2,2)}

```

fillsepX=dim

Default: Opt

The value of the horizontal separation between consecutive patterns.³¹

fillsepY=dim

Default: Opt

The value of the vertical separation between consecutive patterns.³¹

fillsep=dim

Default: Opt

The value of the horizontal and vertical separations between consecutive patterns.³²

These values can be negative, which allow the tiles to overlap.

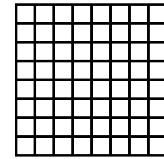
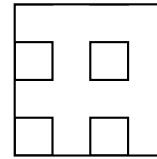
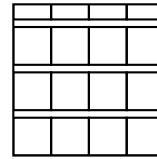
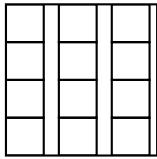
³¹This option is only available in the *tiling* mode.

³²More precisely, the default value is Opt in *tiling* mode and 2pt in *filling* mode.

```

1 \psboxfill{\Square}
2 \Tiling{fillsep=2mm}{(2,2)}\hfill
3 \Tiling{fillsep=1mm}{(2,2)}\hfill
4 \Tiling{fillsep=0.5}{(2,2)}\hfill
5 \Tiling{fillsep=-0.25}{(2,2)}

```



fillclex=*num*

Default: 0

The shift coefficient applied to each row.³¹

fillcley=*num*

Default: 0

The shift coefficient applied to each column.³¹

fillcycle=*num*

Default: 0

The shift coefficient applied both to each row and each column.

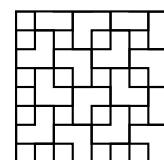
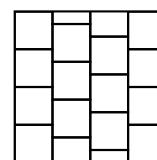
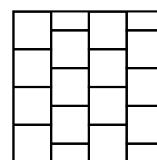
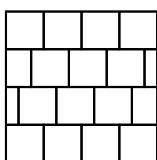
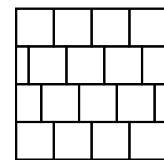
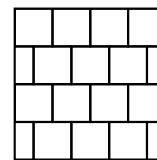
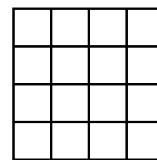
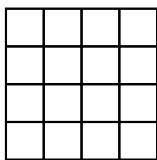
For instance, if **fillclex** is 2, the second row of patterns will be horizontally shifted by a factor of $\frac{1}{2} = 0.5$, and by a factor of 0.333 if **fillclex** is 3, etc.

These values must be integers and can be negative.

```

1 \psboxfill{\Square}
2
3 \Tiling{}{(2,2)}\hfill
4 \Tiling{fillclex=1}{(2,2)}\hfill
5 \Tiling{fillclex=2}{(2,2)}\hfill
6 \Tiling{fillclex=3}{(2,2)}
7
8 \vspace{3mm}
9 \Tiling{fillclex=-3}{(2,2)}\hfill
10 \Tiling{fillcley=2}{(2,2)}\hfill
11 \Tiling{fillcley=-3}{(2,2)}\hfill
12 \Tiling{fillcycle=2}{(2,2)}\hfill

```



fillmovex=dim **Default: Opt**

The value of the horizontal move between consecutive patterns.³¹

fillmovey=dim **Default: Opt**

The value of the vertical move between consecutive patterns.³¹

fillmove=dim **Default: Opt**

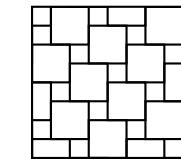
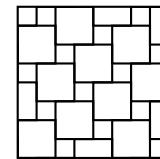
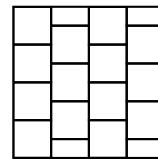
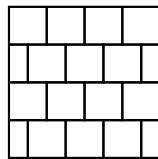
The value of the horizontal and vertical move between consecutive patterns.³¹

These parameters allow the patterns to overlap and to draw some special kinds of tilings.

Their values can be negative.

In some cases, the effect of these parameters will be the same as that with the `fillcycle?` ones, but this is not true for all values.

```
1 \psboxfill{\Square}
2 \Tiling{fillmovex=0.25}{(2,2)}\hfill
3 \Tiling{fillmovey=0.25}{(2,2)}\hfill
4 \Tiling{fillmove=0.25}{(2,2)}\hfill
5 \Tiling{fillmove=-0.25}{(2,2)}
```



fillsize=auto/({(x0,y0)(x1,y1)} **Default: auto**

The choice of *automatic* mode or the size of the area in *manual* mode.

If first pair values are not given, (0,0) is used.

The default value is *auto* in *tiling* mode, (-15cm,-15cm)(15cm,15cm) otherwise.

As explained in the overview, the *manual* mode can use up a large amount of computer resources, so it's usage is therefore discouraged in favour of *automatic* mode, unless special circumstances when some kinds of EPS files are used (see such an example Section 50).

fillloopaddx=num **Default: 0**

The number of times the pattern is added on left and right positions.³¹

fillloopaddy=num **Default: 0**

The number of times the pattern is added on top and bottom positions.³¹

fillloopadd=num**Default: 0**

The number of times the pattern is added on left, right, top and bottom positions.³¹

These parameters (exclusively for the *tiling* mode) are only useful in special circumstances, such as in complex patterns when the size of the rectangular box used to tile the area does not correspond to the pattern itself and also sometimes when the size of the pattern is not a divisor of the size of the area to fill and when the number of loop repeats is not properly computed, which can occur (see such an example Section 50).

PstDebug=num**Default: 0**

To be able to see the exact tiling done, without clipping.³¹

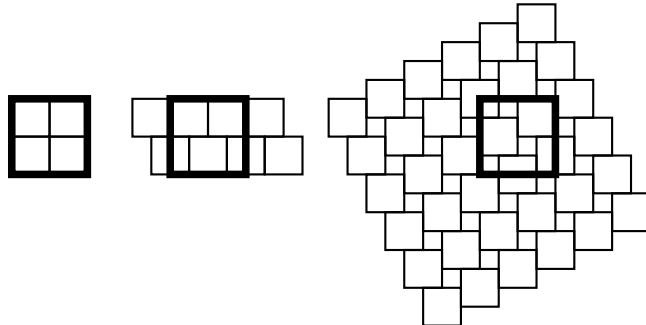
This parameter must be set to 1 to be activated.

This is mainly useful for debugging or to understand better how the tilings are done.

```

1 \psboxfill{\Square}
2 \psset{unit=0.5,dimen=middle,linewidth=1mm,PstDebug=1}
3 \Tiling{(2,2)}\hspace{1cm}
4 \Tiling{fillcyclex=2}{(2,2)}\hspace{3cm}
5 \Tiling{fillmove=0.5}{(2,2)}

```



50 Examples

The single **\psboxfill** macro has many variations and different uses. We will try here to demonstrate many of them:

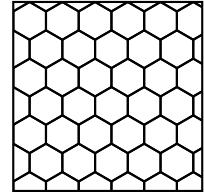
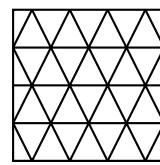
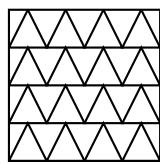
50.1 Kind of tiles

Since we can access all the power of PSTRicks macros to define the *tiles* (*patterns*) used, very complicated ones can be created. Here we give four Archimedean tilings (those built with only some regular polygons) from the eleven known, first discovered completely by Johanes Kepler at the beginning of 17th century, the two *regular* ones with the tiling by squares, formed by a single regular polygon, and two formed by two different regular polygons.

```

1 \def\Triangle{%
2 \pspicture(1,1)\pstriangle[dimen=middle](0.5,0)(1,1)\endpspicture}
3 \def\Hexagon{%
4 \pspicture(0.866,0.75)%  $\sin(60)=0.866$ 
5   \SpecialCoor
6   \pspolygon[dimen=middle](0.5;30)(0.5;90)(0.5;150)
7     (0.5;210)(0.5;270)(0.5;330)%Hexagon
8 \endpspicture}
9
10 \psset{unit=0.5}
11 \psboxfill{\Triangle}
12 \Tiling{(4,4)}\hfill
13 % The two other regular tilings
14 \Tiling{fillcyclex=2}{(4,4)}\hfill
15 \psboxfill{\Hexagon}
16 \Tiling{fillcyclex=2,fillloopaddy=1}{(5,5)}

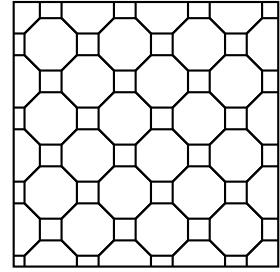
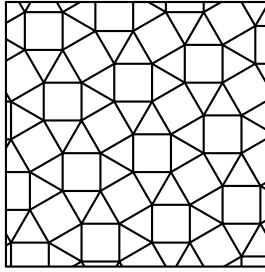
```



```

1 \def\ArchimedianA{%
2 % Archimedean tiling 3^2.4.3.4
3 \psset{dimen=middle}%
4 \pspicture(1.866,1.866)%  $\sin(60)=0.866$ 
5   \psframe(1,1)
6   \psline(1,0)(1.866,0.5)(1,1)
7     (0.5,1.866)(0,1)(-0.866,0.5)
8   \psline(0,0)(0.5,-0.866)
9 \endpspicture}
10
11 \def\ArchimedianB{%
12 % Archimedean tiling 4.8^2
13 \psset{dimen=middle,unit=1.5}%
14 \pspicture(1.3066,0.6533)%  $\cos(22.5) + \sin(22.5) = 1.3066$ 
15 %  $\cos(22.5) - \sin(22.5) = 0.6533$ 
16   \SpecialCoor
17   \pspolygon(0.5;22.5)(0.5;67.5)(0.5;112.5)(0.5;157.5)(0.5;202.5)
18     (0.5;247.5)(0.5;292.5)(0.5;337.5)% Octagon
19 \endpspicture}
20
21 \psset{unit=0.5}
22 \psboxfill{\ArchimedianA}
23 \Tiling{fillmove=0.5}{(7,7)}\hfill
24 \psboxfill{\ArchimedianB}
25 \Tiling{fillcyclex=2,fillloopaddy=1}{(7,7)}

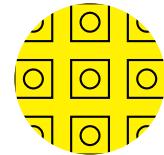
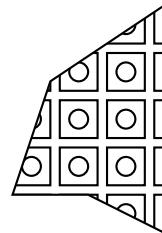
```



We can of course tile an arbitrarily defined area; with the **addfillstyle** parameter, we can easily mix the **boxfill** style with another one.

```

1 \psset{unit=0.5,dimen=middle}
2 \psboxfill{%
3   \pspicture(1,1)
4     \psframe(1,1)
5     \pscircle(0.5,0.5){0.25}
6   \endpspicture}
7
8 \begin{pspicture}(4,6)
9   \pspolygon[fillstyle=boxfill,fillsep=0.25](0,1)(1,4)(4,6)(4,0)(2,1)
10 \end{pspicture}
11 \hspace{2cm}
12 \begin{pspicture}(4,4)
13   \pscircle*[linestyle=none,fillstyle=solid,fillcolor=yellow,fillsep=0.5,
14     addfillstyle=boxfill](2,2){2}
15 \end{pspicture}
```



Various effects can be obtained; sometimes complicated ones are surprisingly easy, as in this example reproduced from one by Slavik Jablan in the field of *OpTiles*, inspired by *Op-art*:

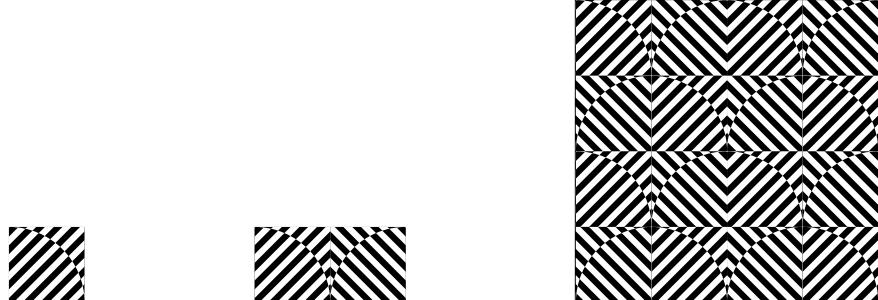
```

1 \def\ProtoTile{%
2 \pspicture(1,1)
3   \psset{linestyle=none,linewidth=0,hatchwidth=0.08333\psunit,
4         hatchsep=0.08333\psunit}%
5   \psframe[fillstyle=solid,fillcolor=black,addfillstyle=hlines,
6         hatchcolor=white](1,1)
7   \pswedge[fillstyle=solid,fillcolor=white,addfillstyle=hlines]{1}{0}{90}
8 \endpspicture}
9
10 \def\BasicTile{%
```

```

11 \pspicture(2,1)
12   \rput[lb](0,0){\ProtoTile}
13   \rput[lb](1,0){\rotateleft{\ProtoTile}}
14 \endpspicture
15
16 \ProtoTile\hfill\BasicTile\hfill
17 \psboxfill{\BasicTile}
18 \Tiling{fillclex=2}{(4,4)}

```

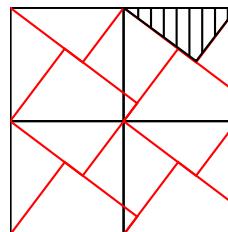


It is also possible to superimpose several different tilings. Here is the splendid visual proof of the Pythagore theorem done by the Arab mathematician Annairizi around the year 900, given by superposition of two tilings by squares of different sizes.

```

1 \psset{unit=1.5,dimen=middle}
2 \pspicture*(2,2)
3   \psboxfill{\pspicture(1,1)
4     \psframe(1,1)
5     \endpspicture}
6   \psframe[fillstyle=boxfill](2,2)
7   \psboxfill{\pspicture(1,1)
8     \rput{-37}{\psframe[linecolor=red](0.8,0.8)}
9     \endpspicture}
10  \psframe[fillstyle=boxfill](3,4)
11  \pspolygon[fillstyle=hlines,hatchangle=90](1,2)(1.64,1.53)(2,2)
12 \endpspicture

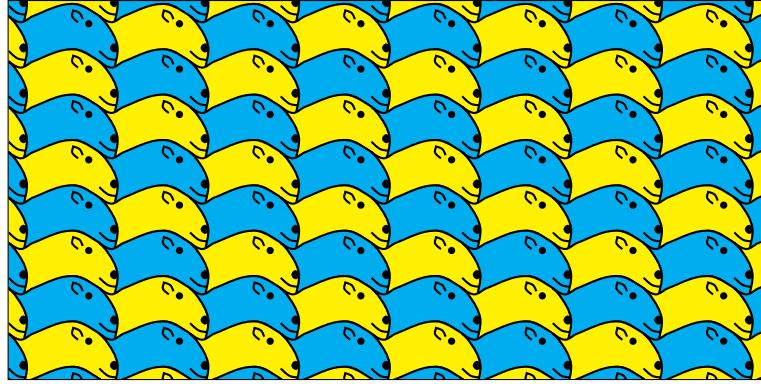
```



In a same way, it is possible to build tilings based on figurative patterns, in the style of the famous Escher ones. Following an example of André Deledicq, the next example shows a simple tiling of the *p1* category (according to the international classification of the 17 symmetry groups of the plane, first discovered by the Russian crystallographer Jevgraf Fedorov at the end of the 19th century).

```

1 \def\SheepHead#1{%
2 \pspicture(3,1.5)
3   \pscustom*[liftpen=2,fillstyle=solid,fillcolor=#1]{%
4     \pscurve(0.5,-0.2)(0.6,0.5)(0.2,1.3)(0,1.5)(0,1.5)(0.4,1.3)(0.8,1.5)
5       (2.2,1.9)(3,1.5)(3,1.5)(3.2,1.3)(3.6,0.5)(3.4,-0.3)(3,0)
6       (2.2,0.4)(0.5,-0.2)}
7     \pscircle*(2.65,1.25){0.12\psunit} % Eye
8     \psccurve*(3.5,0.3)(3.35,0.45)(3.5,0.6)(3.6,0.4)% Muzzle
9     \pscurve(3,0.35)(3.3,0.1)(3.6,0.05) % Mouth
10    \pscurve(2.3,1.3)(2.1,1.5)(2.15,1.7)
11    \pscurve(2.1,1.7)(2.35,1.6)(2.45,1.4) % Ear
12  \endpspicture}
13
14 \psboxfill{\psset{unit=0.4}\SheepHead{yellow}\SheepHead{cyan}}
15 \Tiling{fillclex=2,fillloopadd=1}{(10,5)}
```

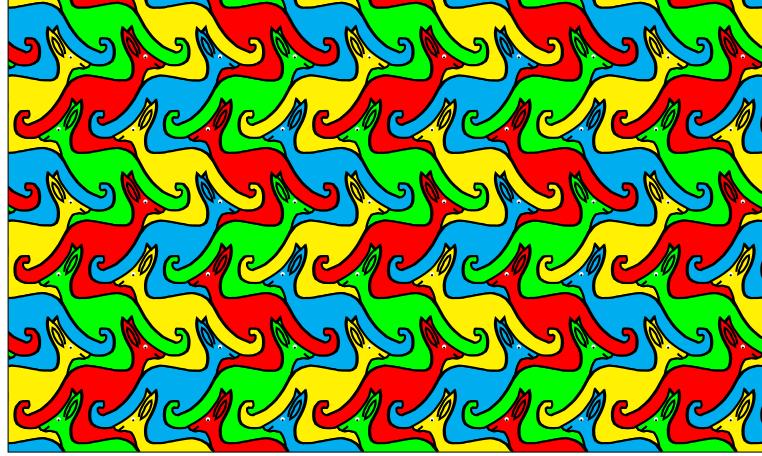


The next example shows a tiling of the *pg* category (the code for the kangaroo itself is too long to be shown here, but has no difficulties).³³; the kangaroo is reproduced from an original picture by Raoul Raba and here is a translation into PStricks from the one drawn by Emmanuel Chailloux and Guy Cousineau for their MLgraph system [6]

```

1 \psboxfill{%
2   \psset{unit=0.4}%
3   \Kangaroo{yellow}\Kangaroo{red}%
4   \Kangaroo{cyan}\Kangaroo{green}%
5   \scalebox{-1 1}{%
6     \rput(1.235,4.8){%
7       \Kangaroo{green}\Kangaroo{cyan}%
8       \Kangaroo{red}\Kangaroo{yellow}}}}%
9 \Tiling{fillloopadd=1}{(10,6)}
```

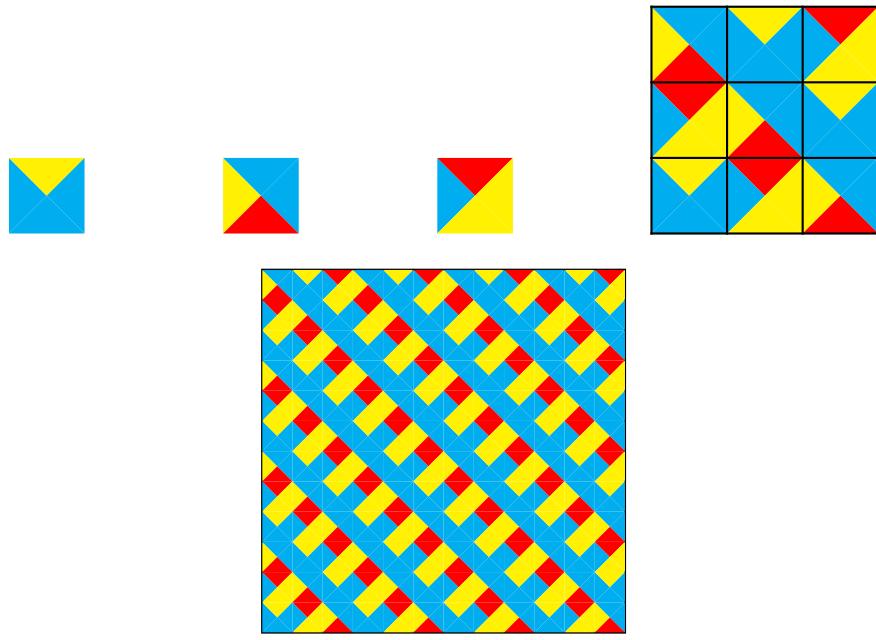
³³You will find it in the source code of this documentation.



And now a Wang tiling [84], based on very simple tiles in the form of a square and composed of four colored triangles. Such tilings are simply built with a matching color constraint. Despite its simplicity, it is an important kind of tiling, as Wang and others used them to study the special class of *aperiodic* tilings, and also because it was shown that (surprisingly) this tiling is similar to a Turing machine.

```

1 \def\WangTile#1#2#3#4{%
2 \pspicture(1,1)
3   \pspolygon*[linecolor=#1](0,0)(0,1)(0.5,0.5)
4   \pspolygon*[linecolor=#2](0,1)(1,1)(0.5,0.5)
5   \pspolygon*[linecolor=#3](1,1)(1,0)(0.5,0.5)
6   \pspolygon*[linecolor=#4](1,0)(0,0)(0.5,0.5)
7 \endpspicture}
8
9 \def\WangTileA{\WangTile{cyan}{yellow}{cyan}{cyan}}
10 \def\WangTileB{\WangTile{yellow}{cyan}{cyan}{red}}
11 \def\WangTileC{\WangTile{cyan}{red}{yellow}{yellow}}
12
13 \def\WangTiles#1{%
14 \pspicture(3,3)
15   \psset{ref=lb}%
16   \rput(0,2){\WangTileB}\rput(1,2){\WangTileA}\rput(2,2){\WangTileC}
17   \rput(0,1){\WangTileC}\rput(1,1){\WangTileB}\rput(2,1){\WangTileA}
18   \rput(0,0){\WangTileA}\rput(1,0){\WangTileC}\rput(2,0){\WangTileB}
19   #1
20 \endpspicture}
21
22 \WangTileA\hfill\WangTileB\hfill\WangTileC\hfill
23 \WangTiles{\psgrid[subgriddiv=0,gridlabels=0](3,3)}
24
25 \vspace{2mm}
26 \psset{unit=0.4}
27 \psboxfill{\WangTiles{}}
28 \Tiling{(12,12)}
```

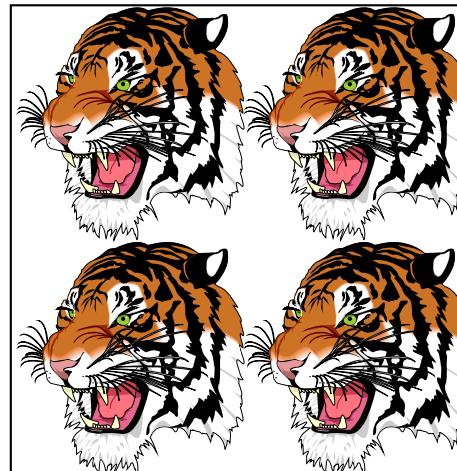


50.2 External graphic files

We can fill an arbitrary area with an external PostScript image. We have only, as usual, to worry about the *BoundingBox* definition if there is not one provided or if it is inaccurate, as in the case of the well known tiger picture (part of the Ghostscript distribution).

```

1 \psboxfill{%
2   \raisebox{-1cm}{%
3     \includegraphics[bb=17 176 562 740,width=3cm]{tiger}}}
4 \Tiling{(6,6.2)}
```



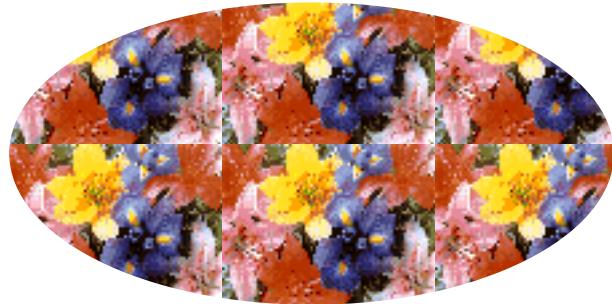
Be warned there are some types of PostScript file for which the *automatic* mode does not work, specifically those produced by a screen dump. This is demonstrated in the next example, where a picture was reduced before

conversion to the *Encapsulated PostScript* format by a screen dump utility. In this case, use of the *manual* mode is the only alternative, at the price of real multiple inclusion of the EPS file. We must take care to specify the correct *fillsize* parameter, because otherwise the default values are large and will load the file too many times, perhaps just actually using a few occurrences as the other ones are clipped away...

```

1 \psboxfill{includegraphics{flowers}}
2 \pspicture(8,4)
3   \psellipse[fillstyle=boxfill,fillsize={(8,4)}](4,2)(4,2)
4 \endpspicture

```



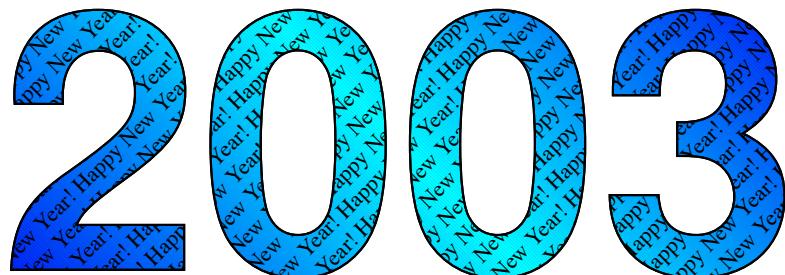
50.3 Tiling of characters

We can also use the **\psboxfill** macro to fill the interior of characters for special effects like the following:

```

1 \DeclareFixedFont{\Sf}{T1}{phv}{b}{n}{5cm}
2 \DeclareFixedFont{\Rm}{T1}{ptm}{m}{n}{3mm}
3 \psboxfill{\Rm Happy New Year!}
4 \pspicture*(10.8,3.8)
5   \rput(5.4,0.1){%
6     \pscharpath[fillstyle=gradient,gradangle=-45,gradmidpoint=0.5,
7       addfillstyle=boxfill,fillangle=45,fillsep=0.7mm]
8       {\rput[b]{0,0}{\Sf 2003}}}
9 \endpspicture

```



```

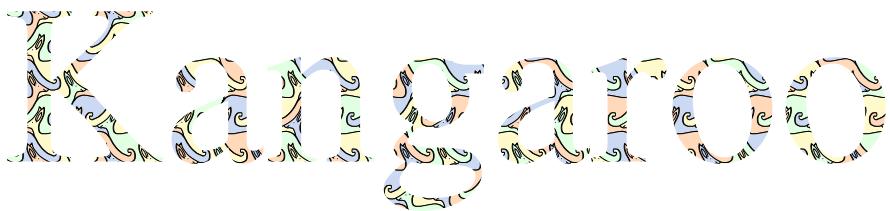
1 \DeclareFixedFont{\Rmm}{T1}{ptm}{m}{n}{3cm}
2 \psboxfill{%
3   \psset{unit=0.2,linewidth=0.2pt}%

```

```

4  \Kangaroo{PeachPuff}\Kangaroo{PaleGreen}%
5  \Kangaroo{LightBlue}\Kangaroo{LemonChiffon}%
6  \scalebox{-1 1}{%
7    \rput(1.235,4.8){%
8      \Kangaroo{LemonChiffon}\Kangaroo{LightBlue}%
9      \Kangaroo{PaleGreen}\Kangaroo{PeachPuff}}}}
10 % A kangaroo of kangaroos...
11 \begin{pspicture}(11.6,2.7)
12   \pscharpath[linestyle=none,fillloopadd=1,fillstyle=boxfill]
13     {\rput[b](5.8,0){\Rmm Kangaroo}}
14 \end{pspicture}

```



50.4 “Dynamic” tiling

In some cases, tilings use *non-static* tiles, that is to say the *prototile(s)*, even if unique, can have several forms, for instance specified by different colors or rotations, not fixed before generation, or varying each time.

We present here as an example the so-called *Truchet* tiling, which is in fact better called *Lewthwaite-Pickover-Truchet (LPT)* tiling.³⁴ The single prototile is just a square with two opposing circle arcs. This tile obviously has two positions, if we rotate it through 90 degrees (see the two tiles on the next figure). A *LPT tiling* is a tiling with randomly oriented LPT tiles. We can see that even if it is very simple in its principle, it draws sophisticated curves with strange properties.

Unfortunately, the ‘pst-fill’ package does not work in a straightforward manner, because the **\psboxfill** macro stores the content of the tile in a TeX box, which is static. So the call of the random function is done only once, which explains why only one rotation of the tile is used for all the tiling. Only one of the two rotations can differ from one drawing to the next...

```

1 \def\ProtoTileLPT{{% LPT prototile
2 \psset{dimen=middle}%
3 \begin{pspicture}(1,1)
4   \psframe(1,1)
5   \psarc(0,0){0.5}{0}{90}
6   \psarc(1,1){0.5}{-180}{-90}
7 \end{pspicture}}

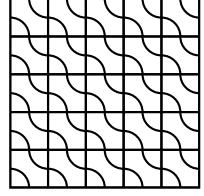
```

³⁴For description of the context, history and references about Sébastien Truchet and this tiling, see the article by Philippe Esperet and Denis Girou in Les Cahiers GUTenberg [13].

```

8  \newcount\Boolean
9
10 \def\BasicTileLPT{%
11   \setrannum{\Boolean}{0}{1}%
12   \ifnum\Boolean=0
13     \ProtoTileLPT%
14   \else
15     \rotateleft{\ProtoTileLPT}%
16   \fi}
17
18 \ProtoTileLPT\hfill
19 \rotateleft{\ProtoTileLPT}\hfill
20 \psset{unit=0.5}%
21 \psboxfill{\BasicTileLPT}
22 \Tiling{(5,5)}
23

```



For simple cases, there is a solution to this problem using a mixture of PStricks and PostScript programming. Here the PStricks construction `\pscustom{\code{...}}` allows us to insert PostScript code inside the L^AT_EX+PStricks one. The programming is less straightforward than to solve this problem using the basic PStricks `\multido` macro, but it has also the advantages of being noticeably faster, since all tilings operations are done in PostScript, and we are not limited by T_EX memory. Note also that `\pslbrace` and `\psrbrace` are PStricks macros which insert the { and } characters.

```

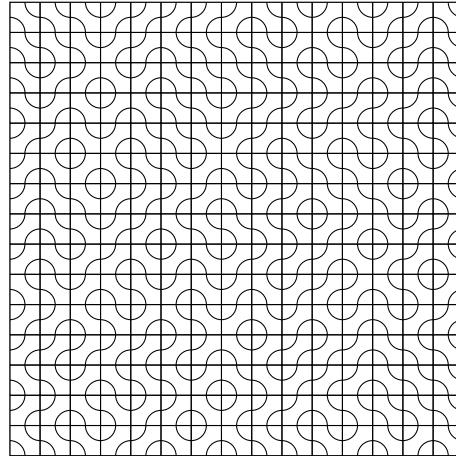
1 \def\ProtoTileLPT{%
2   \psset{dimen=middle}%
3   \psframe(1,1)
4   \psarc(0,0){0.5}{0}{90}
5   \psarc(1,1){0.5}{-180}{-90}}%
6
7 \newcount\InitCounter% Counter to change the random seed
8
9 \def\BasicTileLPT{%
10   \InitCounter=\the\time
11   \pscustom{\code{\rand \the\InitCounter\space sub 2 mod
12     0 eq \pslbrace}}
13   \pspicture(1,1)\ProtoTileLPT\endpspicture%
14   \pscustom{\code{\psrbrace \pslbrace}}
15   \rotateleft{\ProtoTileLPT}%
16   \pscustom{\code{\psrbrace \ifelse}}}
17

```

```

18 \psset{unit=0.4,linewidth=0.4pt}
19 \psboxfill{\BasicTileLPT}
20 \Tiling{(15,15)}

```



Using the very surprising fact that the coloring of these tiles does not depend on their neighbors (even if it is difficult to believe as the opposite seems obvious!) but only on the parity of the value of row and column positions, we can directly program in the same way a colored version of the LPT tiling.

The ‘pst-fill’ package has in *tiling* mode two accessible PostScript variables, *row* and *column*,³³ which can be useful in some circumstances, like this one.

```

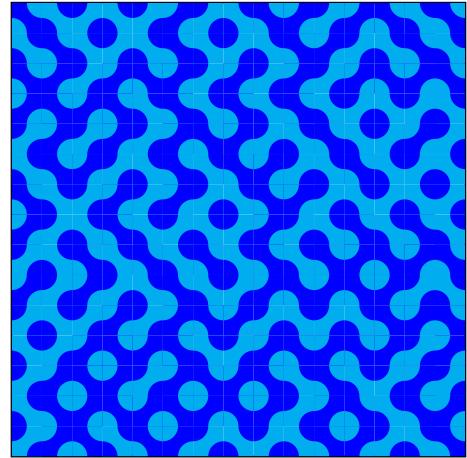
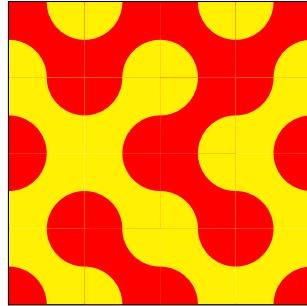
1 \def\ProtoTileLPT#1#2{\% LPT prototile
2 \psset{dimen=middle,linestyle=none,fillstyle=solid}%
3 \psframe[fillcolor=#1](1,1)
4 \psset{fillcolor=#2}%
5 \pswedge(0,0){0.5}{0}{90}
6 \pswedge(1,1){0.5}{-180}{-90}}
7
8 \newcount\InitCounter% Counter to change the random seed
9
10 \def\BasicTileLPT#1#2{\% LPT tile
11 \InitCounter=\the\time
12 \pscustom{code{\rand \the\InitCounter\space sub 2 mod
13 0 eq \pslbrace row column add 2 mod 0 eq \pslbrace}}
14 \pspicture(1,1)\ProtoTileLPT{#1}{#2}\endpspicture%
15 \pscustom{code{\psrbrace \pslbrace}}
16 \ProtoTileLPT{#2}{#1}%
17 \pscustom{code{\psrbrace ifelse \psrbrace \pslbrace
18 row column add 2 mod 0 eq \pslbrace}}
19 \rotateleft{\ProtoTileLPT{#2}{#1}}%
20 \pscustom{code{\psrbrace \pslbrace}}
21 \rotateleft{\ProtoTileLPT{#1}{#2}}%
22 \pscustom{code{\psrbrace ifelse \psrbrace ifelse}}}
23

```

```

24 \psboxfill{\BasicTileLPT{red}{yellow}}
25 \Tiling{(4,4)}
26 \hfill
27 \psset{unit=0.4}
28 \psboxfill{\BasicTileLPT{blue}{cyan}}
29 \Tiling{(15,15)}

```



Another classic example is generation of coordinates and labelling for a grid. Of course, it is possible to do it directly in PStricks using nested `\multido` commands, and it would clearly be easy to program. Nevertheless, for users who have a little knowledge of PostScript programming, this method offers an alternative which is useful for large cases, because it will be noticeably faster and use less computer resources.

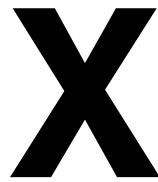
Remember here that the tiling is drawn from left to right, and top to bottom, and note that the PostScript variable `x2` contains the total number of columns.

```

1 % \Escape will be the \ character
2 {\catcode`!=0\catcode`\=11!gdef!\Escape{}}
3
4 \def\ProtoTile{%
5 \pspicture(1,1)\psframe[dimen=middle](1,1)\endpspicture%
6 \pscustom{%
7 \moveto(-0.9,0.75)% In PStricks units
8 \code{/Times-Italic findfont 8 scalefont setfont
9 (\Escape) show row 3 string cvs show (,) show
10 column 3 string cvs show (\Escape)) show}
11 \moveto(-0.5,0.25)% In PStricks units
12 \code{/Times-Bold findfont 18 scalefont setfont
13 1 0 0 setrgbcolor% red color
14 /center {dup stringwidth pop 2 div neg 0 rmoveto} def
15 row 1 sub x2 mul column add 3 string cvs center show}}}
16 \psboxfill{\ProtoTile}
17 \Tiling{(6,4)}

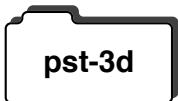
```

$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$	$(1,5)$	$(1,6)$
1	2	3	4	5	6
$(2,1)$	$(2,2)$	$(2,3)$	$(2,4)$	$(2,5)$	$(2,6)$
7	8	9	10	11	12
$(3,1)$	$(3,2)$	$(3,3)$	$(3,4)$	$(3,5)$	$(3,6)$
13	14	15	16	17	18
$(4,1)$	$(4,2)$	$(4,3)$	$(4,4)$	$(4,5)$	$(4,6)$
19	20	21	22	23	24



Three Dimensional Graphics³⁶

51 Overview



The file `pst-3d.tex/pst-3d.sty` contains a macro to add a shadow to some text, two macros to angle objects into the third dimension and a general macro to put objects into three dimensional space. This last command is powerful but supports only parallel (isometric) perspective, and not other kinds of perspective like linear point perspective or spherical perspective,³⁷ nor hidden-line removal or lighting of objects.

52 Usage

`\psshadow[par]{stuff}`

is a macro to draw a text with a shadow.

Shadow

1 `\psshadow{Huge Shadow}`

The **Tshadowangle** parameter specifies the angle of the shadow in degrees (do not use a value of 0 or 180).

Tshadowangle=num

Default: 60

1 `\psshadow[Tshadowangle=120]{Catherine}`

2

3 `\psshadow[Tshadowangle=-45]{March 25}`

Catherine March 25

The **Tshadowsize** parameter specifies the size of the shadow.

Tshadowsize=dim

Default: 1

1 `\psshadow[Tshadowsize=4]{No!}`

2

3 `\psshadow[Tshadowangle=-45,Tshadowsize=0.5]{Lewis Carroll}`

³⁶This chapter is merely an adaptation of the documentation for this package written in 1998 by Manuel Luque.

³⁷For management of these kinds of perspective representations, you can refer to the developments of Manuel Luque.

The **Tshadowcolor** parameter specifies the color of the shadow.

Tshadowcolor=*color*

Default: lightgray

Good morning!

```
1 \psshadow[Tshadowsize=2,Tshadowcolor=red]{Good morning!}
```

Note that the result will be incorrect if you change the color of the text itself from black (then the shadow will be the same color as the text).

\pstilt[*par*]{*angle*}{*stuff*}

is a macro which angles an object into the third dimension (do not use an angle of 0 or 180).

The world is slanted

```
1 \pstilt{45}{The world is slanted}
```

January	250	90
February	320	20
March	200	-40

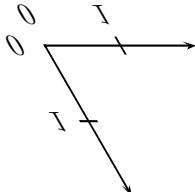
```
1 \pstilt{60}%
2 \begin{tabular}{||lcl|l}
3 & \hline
4 January & 250 & 90 \\ \hline
5 February & 320 & 20 \\ \hline
6 March & 200 & -40 \\ \hline
7 \end{tabular}
```

\psTilt[*par*]{*angle*}{*stuff*}

is a macro which angles an object into the third dimension (do not use an angle of 0 or 180).

The world is slanted

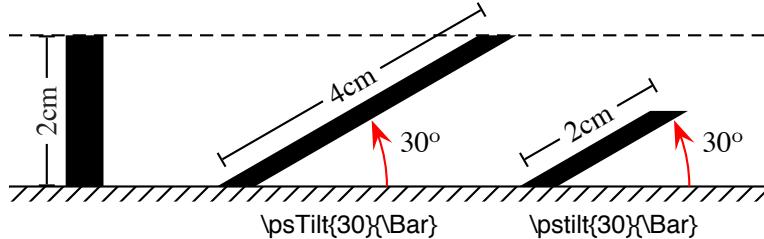
```
1 \psTilt{45}{The world is slanted}
```



```
1 \psTilt{-60}%
2 \begin{pspicture}(2,2)
3 \psaxes(<->)(2,2)
4 \end{pspicture}
```

The difference between **\pstilt** and **\psTilt** is that **\pstilt** keeps the length of the objects constant while **\psTilt** keeps the height of the objects constant as shown in the following example:

```
1 \def\Bar{\psframe*(-0.25,0)(0.25,2)}
2
3 % Additional information is superimposed on the drawing
4 \rput(1,0){\Bar}
5 \rput(3,0){\psTilt{30}{\Bar}}
6 \rput(7,0){\pstilt{30}{\Bar}}
```



These two effects can now easily be understood:

```

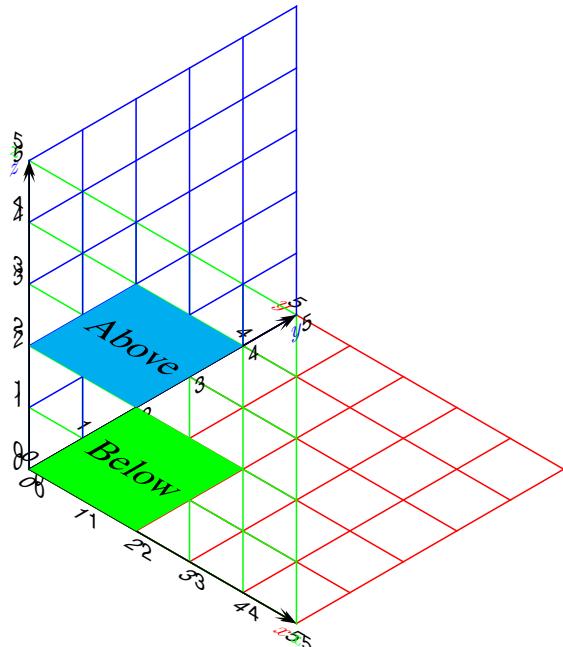
1 \psframebox[framesep=0]{Huge}\sffamily\bfseries%
2 A\psTilt{20}{A}\pstilt{20}{A}}

```



\ThreeDput[par](x0,y0,z0){stuff}

is a generic macro which puts objects into three dimensional space. Everything belonging to a plane can be shown in three dimensional space. If this is not the case, as for a cylinder, a sphere, etc., then it is necessary to decompose the surface into smaller planes of juxtaposed tiles (more planes will produce a more realistic result).



```

1 % The three colored grids are previously drawn
2 \ThreeDput{\psframe*[linecolor=green](2,2)}
3 \ThreeDput(1,1,0){\LARGE Below}
4 \ThreeDput(0,0,2){\psframe*[linecolor=cyan](2,2)}
5 \ThreeDput(1,1,2){\LARGE Above}

```

The **viewpoint** parameter specifies the direction of the observer as shown in the following diagram:

viewpoint= v_x v_y v_z

Default: 1 -1 1

(v_x v_y v_z) are the coordinates of a vector pointing from the origin O to the observers eye. In fact, it defines the direction of the projection

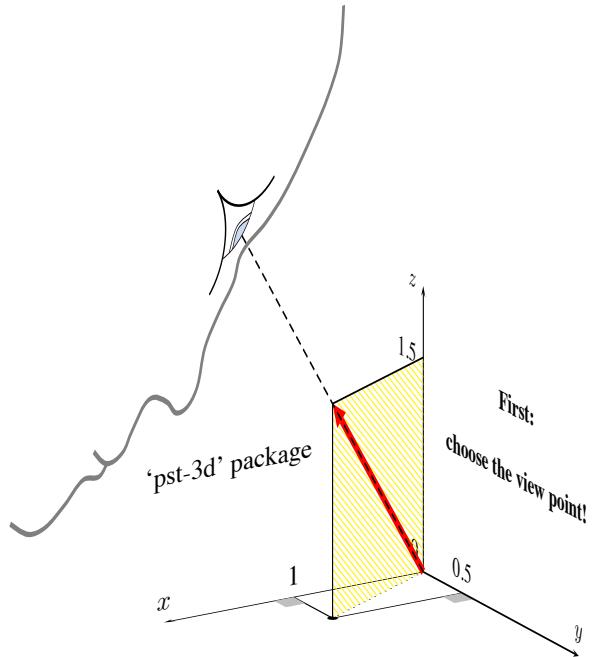


Figure 2: How to choose the viewpoint for the **\ThreeDput** macro.

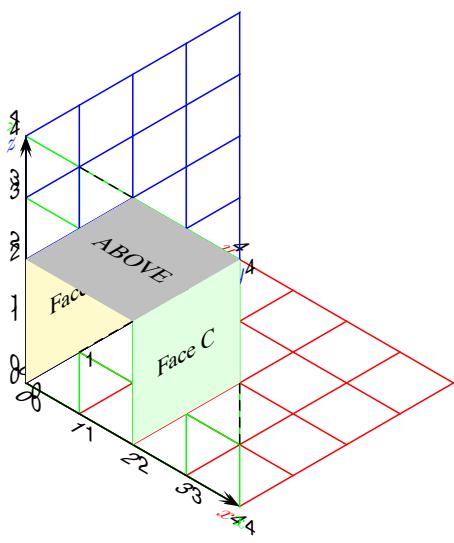
on the horizontal plane. As the vector norm has no importance, it is not necessary to take a unit vector. We must simply choose the coordinates according to the view point that we must have on the object. In figure 2, this vector has $(1\ 0.5\ 1.5)$ for coordinates.³⁸



Warning: Take care to not use one of the coordinates equal to 0, as this will generate a PostScript error. Nevertheless, we can use a very small value, like 0.001 for instance. Other divisions by 0 can occur in very special circumstances, but in this case it is enough to slightly modify the values used.

The cube shown next has its faces drawn in the following order: *Face A* (yellow), *Face B* (blue), *Face C* (green) then face marked *ABOVE* (gray). The various objects are drawn one after the other and there is no management of hidden-line removal. You can see that the order used is very important and must sometimes be changed according to the viewpoint chosen (see the fourth of the following examples).

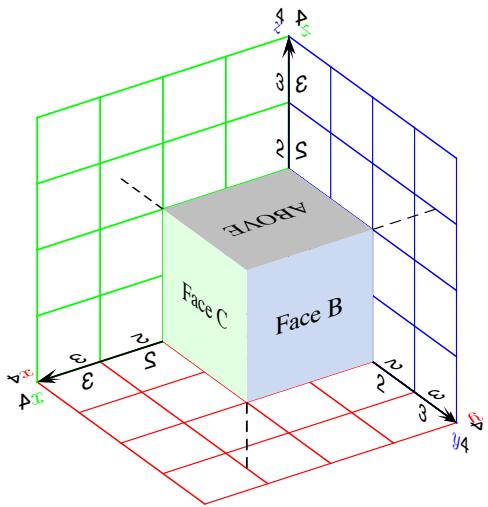
³⁸But the figure itself is not drawn with this value, because we would only see a single point, at the end of the arrow!



```

1 \psset{viewpoint=1 -1 1} % Default value
2 \ShowPartialCube
3 % Note that Face A on the left side is partially covered
4 % and that Face B is totally covered

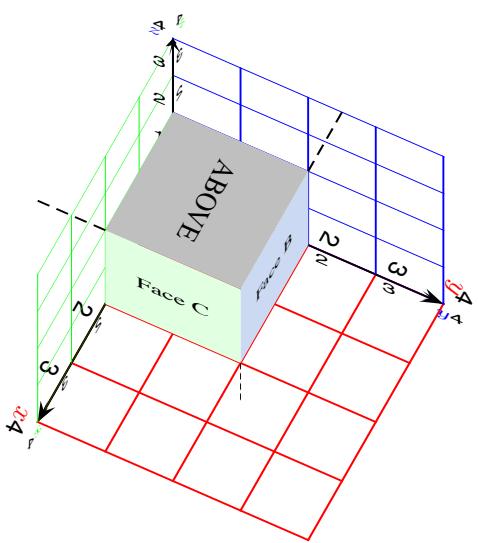
```



```

1 \psset{viewpoint=1 1.5 1}
2 \ShowPartialCube

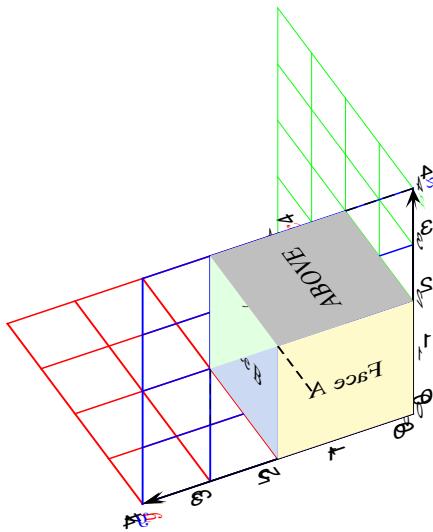
```



```

1 \psset{viewpoint=1 0.5 2}
2 \ShowPartialCube

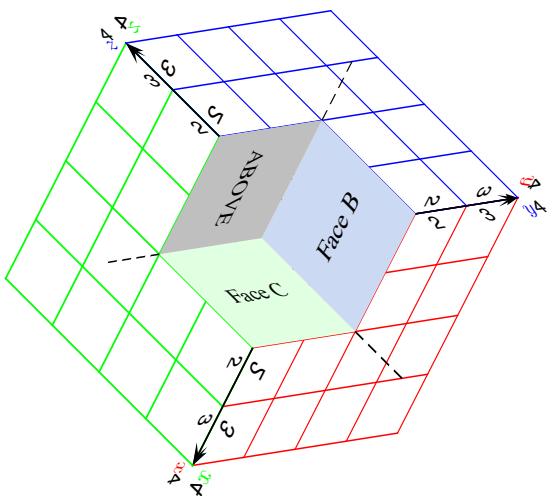
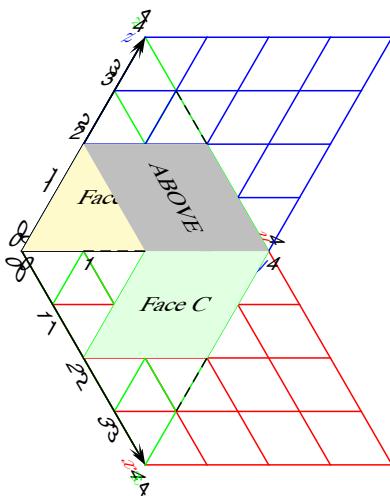
```



The **viewangle** parameter specifies the clockwise rotation applied from the view point.

viewangle=angle

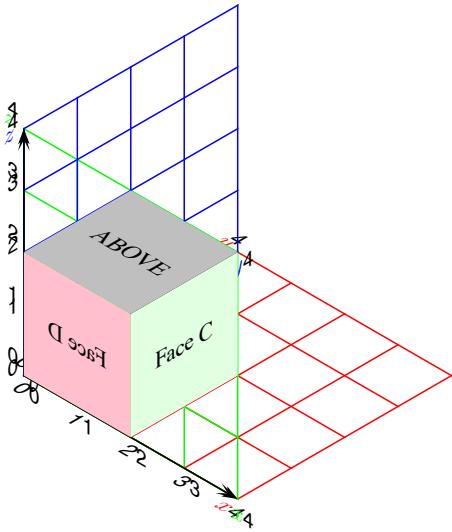
Default: 0



The **normal** parameter specifies the normal vector.

normal= $n_x \ n_y \ n_z$

Default: 0 0 1



```

1 \def\Face#1#2{\psframe*[linecolor=#1](2,2)\rput(1,1){#2}}
2
3 % The three colored grids are previously drawn
4 \ThreeDput[normal=0 1 0](2,0,0){Face}{Face D}
5 \ThreeDput[normal=1 0 0](2,0,0){Face}{PaleGreen}{Face C}
6 \ThreeDput[normal=0 0 1](0,0,2){Face}{lightgray}{ABOVE}

```

We will demonstrate how the normal is to be computed on the example of a rectangle parallelopiped, where we will define only the visible faces from the (default) view point chosen (see the next drawing).

The normal to the face ABCD has the coordinates: $\begin{cases} n_x = 1 \\ n_y = 0 \\ n_z = 0 \end{cases}$

These are the coordinates of a vector which must be perpendicular to the plane that we want to draw. These coordinates are given in the reference landmark $Oxyz$. It is not necessary that this be a unit vector.

The normal to the face ADHE has the coordinates: $\begin{cases} n_x = 0 \\ n_y = 0 \\ n_z = 1 \end{cases}$

The normal to the face AEFB has the coordinates: $\begin{cases} n_x = 0 \\ n_y = -1 \\ n_z = 0 \end{cases}$

etc.

```

1 % The three visible faces of the rectangle parallelopiped
2 % (note that each time we will call the grid macro twice
3 % once for the grid itself and secondly for the labels,
4 % because we do not want to label the extreme values)
5
6 \def\FaceABCD{%
7   \psgrid[gridlabels=0](0,0)(-4,-2)(4,2)
8   \psgrid[gridwidth=0](0,0)(-3,-1)(3,1)
9   \uput[-45]{-4,2}{A}
10  \uput[45]{-4,-2}{B}
11  \uput[135]{4,-2}{C}
}

```

```

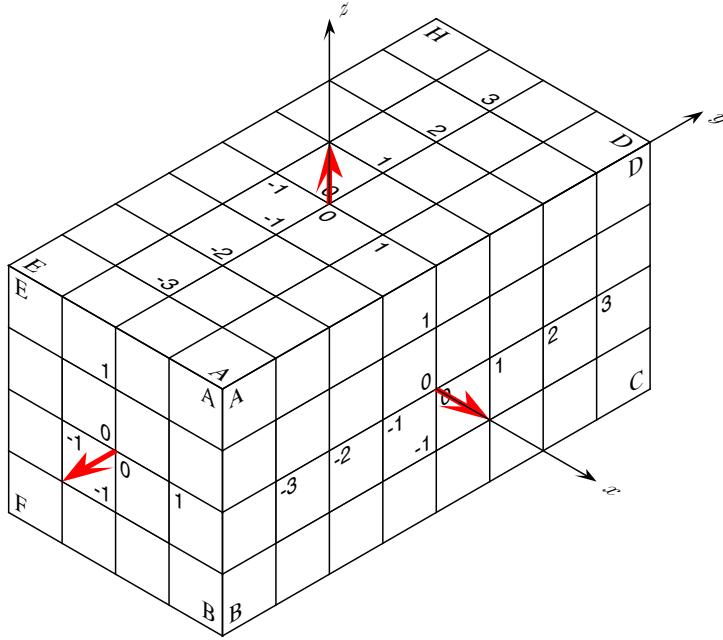
12   \uput[225](4,2){D}}
13
14 \def\FaceADHE{%
15   \psgrid[gridlabels=0](0,0)(-2,-4)(2,4)
16   \psgrid[gridwidth=0](0,0)(-1,-3)(1,3)
17   \uput[135](2,-4){A}
18   \uput[225](2,4){D}
19   \uput[-45](-2,4){H}
20   \uput[45](-2,-4){E}}
21
22 \def\FaceABFE{%
23   \psgrid[gridlabels=0](0,0)(-2,-2)(2,2)
24   \psgrid[gridwidth=0](0,0)(-1,-1)(1,1)
25   \uput[225](2,2){A}
26   \uput[135](2,-2){B}
27   \uput[45](-2,-2){F}
28   \uput[-45](-2,2){E}}
29
30 \pspicture(-4.2,-4.1)(5.3,4.3)
31   \psset{dimen=middle,subgriddiv=0,arrows=>,arrowscale=2}%
32   %
33   % Face ABCD
34   \ThreeDput[normal=1 0 0](2,0,0){\FaceABCD}
35   %
36   % Normal for the plane ABCD.
37   % In its center, we draw the plane where it is placed.
38   % This is the plane parallel to Oxy placed in (2,0,0)
39   % for which the normal is parallel to Oz.
40   \ThreeDput[normal=0 0 1](2,0,0){%
41     \psline[linecolor=red,linewidth=0.1](1,0)
42     \psline(3,0)\uput[0](3,0){$x\$}}
43   %
44   % Face ADHE
45   \ThreeDput[normal=0 0 1](0,0,2){\FaceADHE}
46   %
47   % Normal for the plane ADHE.
48   % In its center, we draw the plane where it is placed.
49   % This is the plane parallel to Oyz placed in (0,0,2)
50   % for which the normal is parallel to Ox.
51   \ThreeDput[normal=1 0 0](0,0,2){%
52     \psline[linecolor=red,linewidth=0.1](0,1)
53     \psline(0,3)\uput[0](0,3){$z\$}}
54   %
55   % Face ABFE
56   \ThreeDput[normal=0 -1 0](0,-4,0){\FaceABFE}
57   %
58   % Normal for the plane ABFE.
59   % In its center, we draw the plane where it is placed.
60   % This is the plane parallel to Oxy placed in (0,-4,0)
61   % for which the normal is parallel to Oy.
62   \ThreeDput[normal=0 0 -1](0,-4,0){%
63     \psline[linecolor=red,linewidth=0.1](0,1)}
64   %

```

```

65 % Oy axis
66 \ThreeDput[normal=0 0 1](0,4,0){%
67   \psline(0,3)\uput[0](0,3){$y$}}
68 \endpspicture

```



Let us suppose that we wish to draw a chamfered edge according to the edge AD. The normal for the chamfer has the coordinates:

$$\begin{cases} n_x = 1 \\ n_y = 0 \\ n_z = 1 \end{cases}$$

and the origin of this plane has the coordinates: $\begin{cases} x_0 = 1.5 \\ y_0 = 0 \\ z_0 = 1.5 \end{cases}$

The dimensions of this chamfered edge are $L = 8$, $l = \sqrt{2} = 1.414$. We will draw the chamfer with its normal.

```

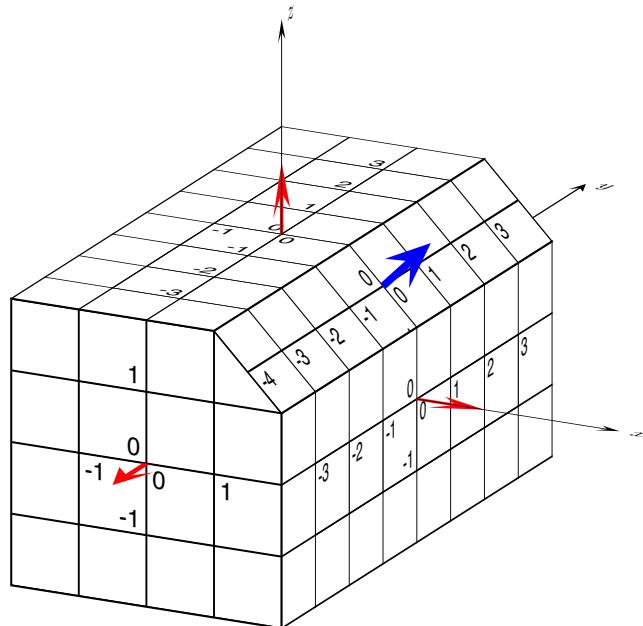
1 \pspicture(-3.6,-3.4)(4.8,4.9)
2 \psset{dimen=middle,subgriddiv=0,arrows=>,arrowscale=2,
3   viewpoint=1 -2 0.75}%
4 %
5 % Oy axis
6 \ThreeDput[normal=0 0 1](0,4,0){%
7   \psline(0,5)\uput[0](0,5){$y$}}
8 %
9 \ThreeDput[normal=1 0 0](2,0,0){%
10   \psgrid[gridlabels=0](0,0)(-4,-2)(4,1)
11   \psgrid[gridwidth=0](0,0)(-3,-1)(3,0)}
12 %
13 \ThreeDput[normal=0 0 1](2,0,0){%
14   \psline[linecolor=red,linewidth=0.1](1,0)
15   \psline(3,0)\uput[0](3,0){$x$}}
16 %
17 \ThreeDput[normal=0 0 1](0,0,2){%
18   \psgrid[gridlabels=0](0,0)(-2,-4)(1,4)
19   \psgrid[gridwidth=0](0,0)(-1,-3)(1,3)}

```

```

20 %
21 \ThreeDput[normal=1 0 0](0,0,2){%
22   \psline[linecolor=red,linewidth=0.1](0,1)
23   \psline(0,3)\put[0](0,3){$z$}
24 %
25 \ThreeDput[normal=0 -1 0](0,-4,0){%
26   \psclip{\pspolygon(-2,-2)(-2,2)(1,2)(2,1)(2,-2)}
27     \psgrid[gridlabels=0](0,0)(-2,-2)(2,2)
28   \endpsclip
29 %
30 \ThreeDput[normal=0 0 -1](0,-4,0){%
31   \psline[linecolor=red,linewidth=0.1](0,1)}
32 %
33 % The chamfered edge
34 \ThreeDput[normal=1 0 1](1.5,0,1.5){%
35   \psclip{\psframe[fillstyle=solid](-4,-0.707)(4,0.707)}
36     \psgrid(0,0)(-4,-0.707)(4,0.707)
37   \endpsclip
38 %
39 % The normal vector to this chamfer
40 \ThreeDput[normal=0 1 0](1.5,0,1.5){%
41   \psline[linecolor=blue,linewidth=0.1](-0.707,0.707)}
42 \endpspicture

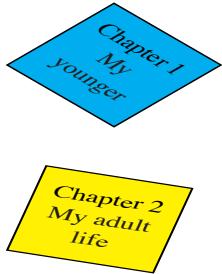
```



The **embedangle** parameter rotates the object about the normal in a counter clockwise (if **embedangle > 0**) direction.

embedangle=angle

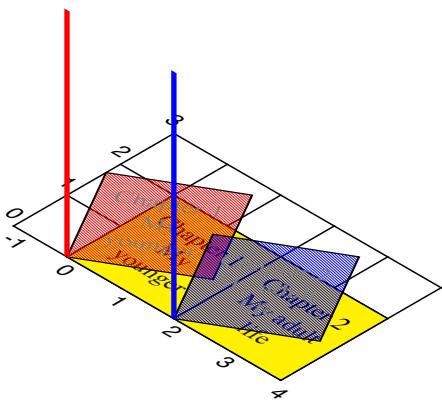
Default: 0



```

1 \def\BookChapter#1#2#3#4{%
2   \psframe[style=#4](2,2)
3   \rput(1,1){%
4     \begin{minipage}{1.5cm}
5       \centering
6       \textcolor{#3}{Chapter #1} \\
7       #2
8     \end{minipage}}
9
10 \newpsstyle{SolidCyan}{fillstyle=solid,fillcolor=cyan}
11 \newpsstyle{SolidYellow}{fillstyle=solid,fillcolor=yellow}
12
13 \ThreeDput(-2,-2,0){%
14   \BookChapter{1}{My younger}{black}{SolidCyan}}
15
16 \ThreeDput[embedangle=30](-2,-2,-3){%
17   \BookChapter{2}{My adult life}{black}{SolidYellow}}

```



```

1 \newpsstyle{TransparencyRed}{fillstyle=vlines,hatchcolor=red,
2   hatchwidth=0.1\pslinewidth,hatchsep=1\pslinewidth}
3 \newpsstyle{TransparencyBlue}{fillstyle=vlines,hatchcolor=blue,
4   hatchwidth=0.1\pslinewidth,hatchsep=1\pslinewidth}
5
6 % Underlaying grid
7 \ThreeDput{\psgrid[subgriddiv=0](-1,0)(4,3)}
8
9 % embedangle=0
10 \ThreeDput{%
11   \BookChapter{1}{My younger}{black}{SolidYellow}}
12 \ThreeDput(2,0,0){%
13   \BookChapter{2}{My adult life}{black}{SolidYellow}}
14
15 \bgroup
16   \psset{embedangle=30}%
17   % embedangle=30
18   \ThreeDput{%
19     \BookChapter{1}{\textcolor{gray}{My younger}}%
20     {\textcolor{gray}{TransparencyRed}}}
21   \ThreeDput(2,0,0){%
22     \BookChapter{2}{\textcolor{gray}{My adult life}}%
23     {\textcolor{gray}{TransparencyBlue}}}
24 \egroup
25
26 % Normals
27 \ThreeDput[normal=0 1 0]{%
28   \psline[linewidth=0.1,linecolor=red](0,4)}
29 \ThreeDput[normal=0 1 0](2,0,0){%
30   \psline[linewidth=0.1,linecolor=blue](0,4)}

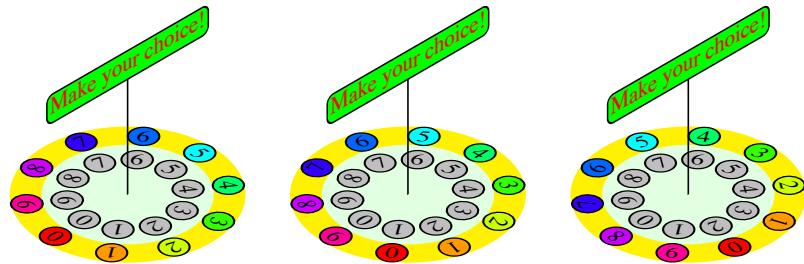
```

It is not really possible to use this option on complex scenes, with composite objects, because the behavior is not equivalent to applying a global change. This is especially important when a scene includes

objects with different normals, because if each object turns around its normal by the same angle, this is not equivalent to turning the whole composite object, as illustrated in the next example.

```

1 \psset{unit=0.3,dimen=middle,framesep=0.15,viewpoint=1 1 1}
2 \SpecialCoor
3
4 \multido{\iEmbedAngle=0+36}{3}{%
5   \pspicture(-5.2,-3)(5.2,8.3)
6     \ThreeDput{%
7       \pscircle*[linecolor=PaleGreen]{4.5}
8       \multido{\iNum=0+1,\iAngle=0+36,\iRot=-90+36}{10}{%
9         \rput{\iRot}{(2.7;\iAngle)}{%
10           \pscirclebox[fillstyle=solid,fillcolor=lightgray]{%
11             \small\iNum}}}
12     \ThreeDput[embedangle=\iEmbedAngle]{%
13       \pscircle[linewidth=1.4,linecolor=yellow]{4.5}
14       \multido{\iNum=0+1,\iAngle=0+36,\iRot=-90+36,
15         \nHue=0.0+0.1}{10}{%
16         \definecolor{ColorNumber}{hsb}{\nHue,1,1}}
17         \rput{\iRot}{(4.5;\iAngle)}{%
18           \pscirclebox[fillstyle=solid,fillcolor=ColorNumber]{%
19             \small\iNum}}}
20     \ThreeDput[normal=0 1 0]{%
21       \psline[linewidth=0.1](0,7)
22       \rput(0,7){%
23         \psframebox[fillstyle=solid,fillcolor=green,framearc=.5]{%
24           \textcolor{red}{\text{Make your choice!}}}}}
25   \endpspicture
26 }
```

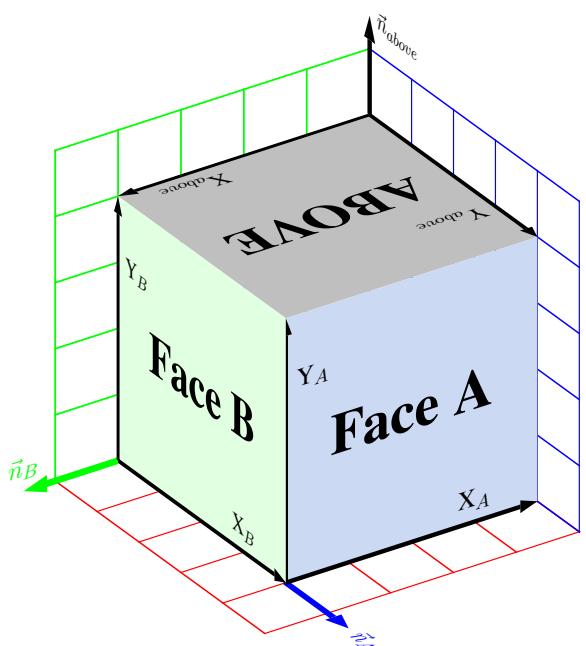


53 More details

The macro **\ThreeDput**[normal= $n_x \ n_y \ n_z$]($\Omega_x \ \Omega_y \ \Omega_z$) allows us to mathematically define the plane by a vector normal to this plane and by a point Ω belonging to this plane. This point Ω is considered as the origin of the axes system (ΩXY), on which all the positioning of the points on this plane are entered. This landmark is automatically defined by the **\ThreeDput** macro.

In the next drawing, the normal vectors to the three faces are shown with both of their associated landmarks. It is important to notice the continuity

of the orientation of the faces if we turn the cube around. Also, if we apply a three dimensional rotation to the upper face of the cube so that the X_{above} axis is parallel to the X_B axis of face B, we can see that the two faces are aligned.



```

1 \psset{viewpoint=1 1.5 1,arrows=->,arrowinset=0,
2         gridlabels=0,subgriddiv=0}
3 % Grids
4 \ThreeDput[normal=0 0 1]{% Oxy plane
5   \psgrid[gridcolor=red](5,5)
6   \psline[linewidth=3pt,linestyle=blue](4,4)(4,5.5)
7   \uput[90](4,5.5){%
8     \rotateleft{\textcolor{blue}{$\vec{n}_A$}}}
9 \ThreeDput[normal=0 -1 0]{% Oxz plane
10  \psgrid[gridcolor=green](5,5)
11  \psline[linewidth=3pt,linestyle=green](4,0)(5.5,0)
12  \uput[90](5.5,0){%
13    \scalebox{-1 1}{\textcolor{green}{$\vec{n}_B$}}}
14 \ThreeDput[normal=1 0 0]{% Oyz plane
15  \psgrid[gridcolor=blue](5,5)
16  \psline[linewidth=3pt](0,4)(0,5.5)
17  \uput[0](0,5.5){$\vec{n}_A$}
18 % Cube and axes
19 \psset{linewidth=2pt}
20 \ThreeDput[normal=0 0 1](0,0,4){%
21   \psframe*[linestyle=lightgray](4,4)
22   \rput(2,2){\Huge\textbf{ABOVE}}
23   \psline(4,0)\uput[90](3,0){X$_{above}$}
24   \psline(0,4)\uput[0](0,3){Y$_{above}$}
25 \ThreeDput[normal=0 1 0](4,4,0){%
26   \psframe*[linestyle=LightBlue](4,4)
27   \rput(2,2){\Huge\textbf{Face A}}
28   \psline(4,0)\uput[90](3,0){X$_{A}$}
29   \psline(0,4)\uput[0](0,3){Y$_{A}$}
30 \ThreeDput[normal=1 0 0](4,0,0){%
31   \psframe*[linestyle=PaleGreen](4,4)
32   \rput(2,2){\Huge\textbf{Face B}}
33   \psline(4,0)\uput[90](3,0){X$_{B}$}
34   \psline(0,4)\uput[0](0,3){Y$_{B}$}}

```

What are the orientation rules? Take for instance the face B. If Ω is the point of coordinates $(1,0,0)$ in the landmark $Oxyz$, then $\Omega X_B, \Omega Y_B, \vec{n}_B$ produces a direct trihedral. From a practical point of view, we must have (the three fingers of your right hand producing a trihedral):

- thumb = ΩX_B
- forefinger = ΩY_B
- long finger = \vec{n}_B

All the faces obey to this rule. Imagine standing on a face of the cube with

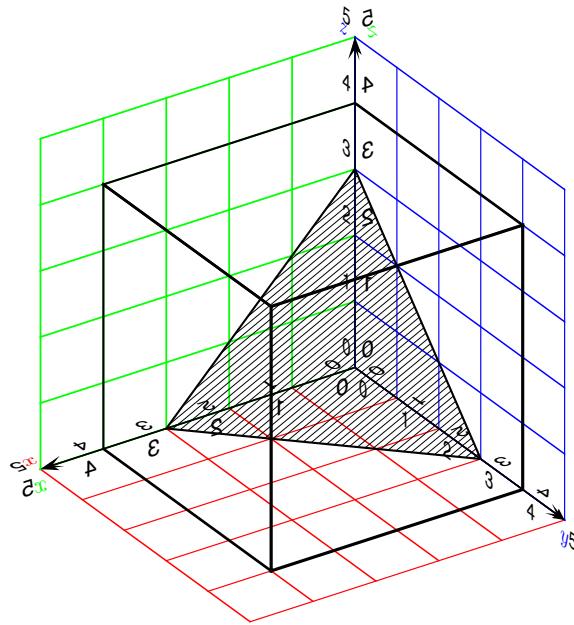
the heels of your feet meeting at one corner, your right foot will be facing Ox , your left foot will be facing Oy and you will be standing in the \vec{n} (the normal) direction. You can write onto the face in the direction that your right foot points toward.

How to represent the intersection of the unit cube with the plane h defined by:

- $0 \leq x \leq 4$
- $0 \leq y \leq 4$
- $0 \leq z \leq 4$

where $h = x + y + z$ with $0 < h < 12$? We will define this plane by its normal and the origin Ω of the new landmark in this plane. A plane of Cartesian equation $ax + by + cz = d$ owns $\vec{n}(a, b, c)$ as its normal vector. In this case, we can then take $\vec{n}(1, 1, 1)$, and easily verify that the point $\Omega(\frac{h}{3}, \frac{h}{3}, \frac{h}{3})$ is a point of this plane. This is the *foot* of the perpendicular of O to this plane. And we can define this plane by the macro
\ThreeDput[normal=1 1 1]($\frac{h}{3}, \frac{h}{3}, \frac{h}{3}$)

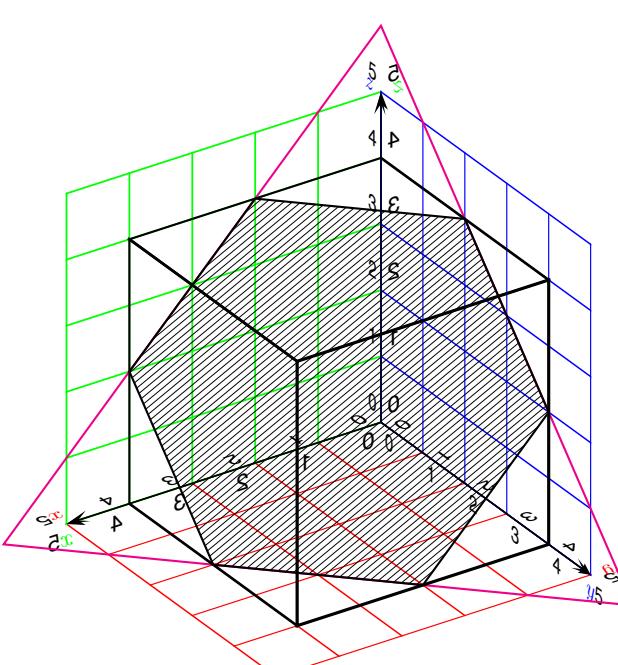
The next examples are related respectively to $h=3$, $h=6$ (regular hexagon) and $h=7$. But, first, we must compute the coordinates of the vertex of the polygons (triangle or hexagon), intersections of the plane with the edges of the cube (this package does not do it, but we can easily code the computations in PostScript). After that we can draw the polygon by the usual **\pspolygon** macro.



```

1 \SpecialCoor
2 \psset{viewpoint=1 1.5 1,arrows=->,arrowscale=2,subgriddiv=0}
3 % Grids and axes
4 \ThreeDput[normal=0 0 1]{% Oxy plane
5   \psgrid[gridcolor=red](5,5)
6   \psline(0,5)\uput[180](0,5){\textcolor{red}{$y$}}
7   \psline(5,0)\uput[-90](5,0){\textcolor{red}{$x$}}}
8 \ThreeDput[normal=0 -1 0]{% Oxz plane
9   \psgrid[gridcolor=green](5,5)
10  \psline(0,5)\uput[180](0,5){\textcolor{green}{$z$}}
11  \psline(5,0)\uput[-90](5,0){\textcolor{green}{$x$}}}
12 \ThreeDput[normal=1 0 0](0,0,0){% Oyz plane
13   \psgrid[gridcolor=blue](5,5)
14   \psline(0,5)\uput[180](0,5){\textcolor{blue}{$z$}}
15   \psline(5,0)\uput[-90](5,0){\textcolor{blue}{$y$}}}
16 % Intersection of the plane x+y+z=h with the cube for h=3
17 % Radius = h*sqrt(2/3) = 3*0.8465 = 2.4495
18 \ThreeDput[normal=1 1 1](1,1,1){% (h/3,h/3,h/3)
19   \pscustom[fillstyle=hlines,hatchwidth=0.1pt,hatchsep=2pt]{%
20     \code{/h 3 def /Radius h 2 3 div sqrt mul def}%
21     \pspolygon(! Radius dup -30 cos mul exch -30 sin mul)
22       (! Radius dup 90 cos mul exch 90 sin mul)
23       (! Radius dup 210 cos mul exch 210 sin mul)}}
24 % Cube
25 \psset{linewidth=0.05}
26 \ThreeDput[normal=0 0 1](0,0,4){\psframe(4,4)}
27 \ThreeDput[normal=0 1 0](4,4,0){\psframe(4,4)}
28 \ThreeDput[normal=1 0 0](4,0,0){\psframe(4,4)}

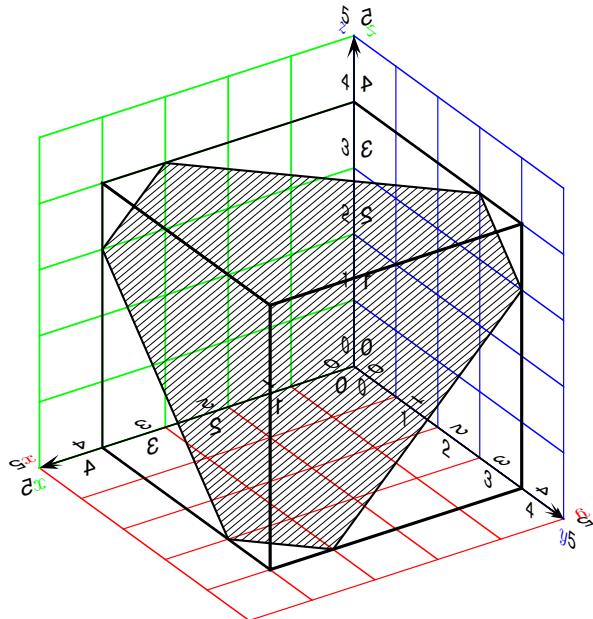
```



```

1 \SpecialCoor
2 \psset{viewpoint=1 1.5 1,arrows=->,arrowscale=2,subgriddiv=0}
3 % Grids and axes
4 % ...
5 % Polygons
6 \ThreeDput[normal=1 1 1](2,2,2){% (h/3,h/3,h/3)
7   % Triangle
8   \pscustom[linecolor=magenta]{%
9     \code{/h 6 def /Radius h 2 3 div sqrt mul def}%
10    \pspolygon(! Radius dup -30 cos mul exch -30 sin mul)
11      (! Radius dup 90 cos mul exch 90 sin mul)
12      (! Radius dup 210 cos mul exch 210 sin mul)}}
13 % Intersection of the plane x+y+z=h with the cube for h=6
14 % Radius = h/3*sqrt(2) = 6/3*1.414 = 2.828
15 \pscustom[fillstyle=hlines,hatchwidth=0.1pt,hatchsep=2pt]{%
16   \code{/h 6 def /Radius h 3 div 2 sqrt mul def}%
17   \pspolygon(! Radius dup 0 cos mul exch 0 sin mul)
18     (! Radius dup 60 cos mul exch 60 sin mul)
19     (! Radius dup 120 cos mul exch 120 sin mul)
20     (! Radius dup 180 cos mul exch 180 sin mul)
21     (! Radius dup 240 cos mul exch 240 sin mul)
22     (! Radius dup 300 cos mul exch 300 sin mul)}}
23 % Cube
24 % ...

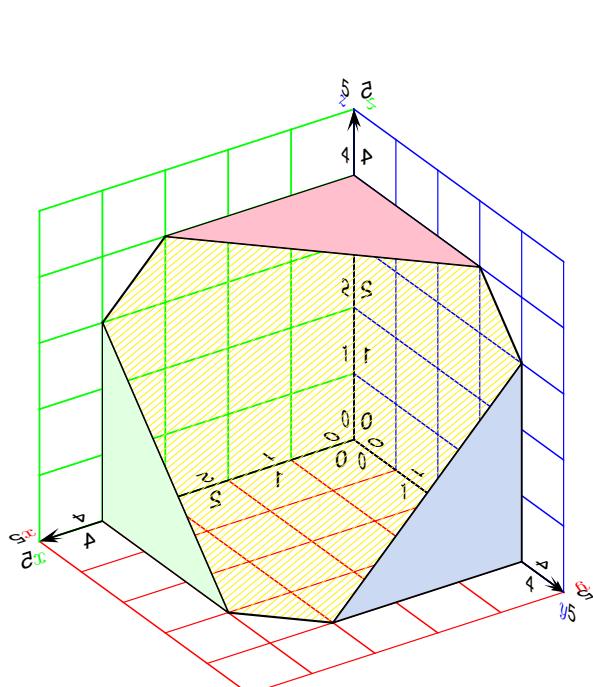
```



```

1 \SpecialCoor
2 \psset{viewpoint=1 1.5 1,arrows=->,arrowscale=2,subgriddiv=0}
3 % Grids and axes
4 % ...
5 % Intersection of the plane  $x+y+z=h$  with the cube for  $h=7$ 
6 \def\Radius{2.9439}%
7 \ThreeDput[normal=1 1 1](2.33,2.33,2.33){% ( $h/3,h/3,h/3$ )
8   \pspolygon[fillstyle=hlines,hatchwidth=0.1pt,hatchsep=2pt]
9     (\Radius;43.9)(\Radius;136.1)(\Radius;163.9)
10    (\Radius;256.1)(\Radius;283.9)(\Radius;376.1)}
11 % Cube
12 % ...

```



```

1 \SpecialCoor
2 \psset{viewpoint=1 1.5 1,arrows=->,arrowscale=2,subgriddiv=0}
3 % Grids and axes
4 % ...
5 % Intersection of the plane  $x+y+z=h$  with the cube for  $h=7$ 
6 \def\Radius{2.9439}%
7 \ThreeDput[normal=1 1 1](2.33,2.33,2.33){% ( $h/3,h/3,h/3$ )
8   \pspolygon[fillstyle=hlines,hatchwidth=0.1pt,hatchsep=2pt,
9     hatchcolor=yellow]
10    (\Radius;43.9)(\Radius;136.1)(\Radius;163.9)
11    (\Radius;256.1)(\Radius;283.9)(\Radius;376.1)}
12 %
13 % On the face A, this is a triangle with edges
14 % of coordinates (4,0)(1,0)(4,3)
15 \ThreeDput[normal=0 1 0](4,4,0){%
16   \pspolygon[fillstyle=solid,fillcolor=LightBlue](4,0)(1,0)(4,3)}
17 % On the face B, this is a triangle with edges
18 % of coordinates (0,0)(0,3)(3,0)
19 \ThreeDput[normal=1 0 0](4,0,0){%
20   \pspolygon[fillstyle=solid,fillcolor=PaleGreen](0,0)(0,3)(3,0)}
21 % On the upper face, this is a triangle with edges
22 % of coordinates (0,0)(3,0)(0,3)
23 \ThreeDput[normal=0 0 1](0,0,4){%
24   \pspolygon[fillstyle=solid,fillcolor=Pink](0,0)(3,0)(0,3)}

```

XI

Other features

54 Special coordinates

The command

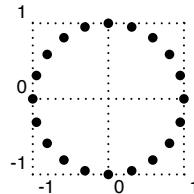
`\SpecialCoor`

enables a special feature that lets you specify coordinates in a variety of ways, in addition to the usual Cartesian coordinates. Processing is slightly slower and less robust, which is why this feature is available on demand rather than by default, but you probably won't notice the difference.

Here are the coordinates you can use:

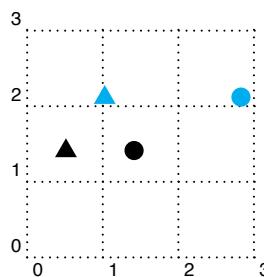
(*x,y*) The usual Cartesian coordinate. E.g., (3,4).

(*r;a*) Polar coordinate, with radius *r* and angle *a*. The default unit for *r* is **unit**. E.g., (3;110).



```
1 \SpecialCoor  
2 \multido{\iAngle=0+18}{20}{\psdot(1,\iAngle)}
```

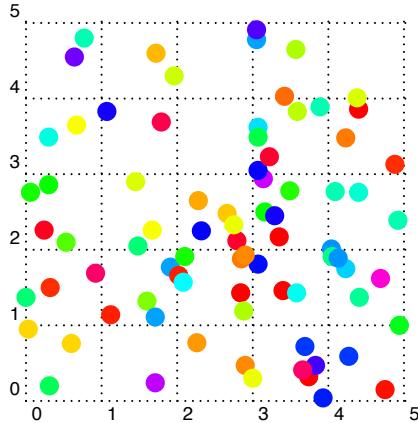
(!*ps*) Raw PostScript code. *ps* should expand to a coordinate pair. The units **xunit** and **yunit** are used. For example, if we want to use polar coordinates (2, 45) and (1.5, 70) that are scaled along with **xunit** and **yunit**, we can write:



```
1 \SpecialCoor  
2 \psset{dotscale=2,xunit=2,yunit=1.5)%  
3 \psdot(2;45)  
4 \psdot[linecolor=cyan](! 2 45 cos mul 2 45 sin mul)  
5 \psset{dotstyle=triangle*}%  
6 \psdot(1.5;70)  
7 \psdot[linecolor=cyan](! 1.5 70 cos mul 1.5 70 sin mul)
```

A lot of things can be done with some PostScript programming, and sometimes in an easiest way than at the TeX level. Note also that some PostScript code can be executed in addition to the computation

of the coordinates. In the next example, eighty dots of random colors (using the HSB color model) are put at random coordinates, in the square $(5,5)$:³⁹

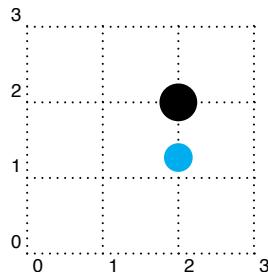


```

1 \SpecialCoor
2 \psset{dotscale=2}%
3 \multips(0,0){80}{%
4   \psdot(! rand 501 mod 100 div rand 501 mod 100 div
5     rand 101 mod 100 div 1 1 sethsbcolor)}

```

(coor1|coor2) The x coordinate from *coor1* and the y coordinate from *coor2*. *coor1* and *coor2* can be any other coordinates for use with **\SpecialCoor**.



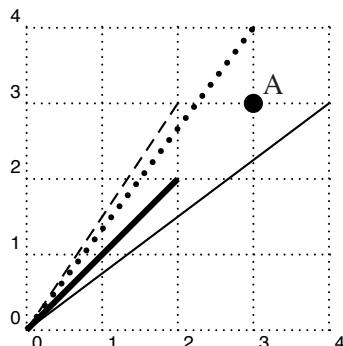
```

1 \SpecialCoor
2 \Cnode*(2,2){A}
3 \psdot[linecolor=cyan,dotscale=3](A1in;30)

```

(node) The center of *node*. E.g., (A).

([par]node) The position relative to *node* determined using the **angle**, **nodesep** and **offset** optional parameters. E.g., ([angle=45]A). Using both **nodesep** and **offset** allow to define relative moves (**node-sep** for horizontal and **offset** for vertical) from an existing node.

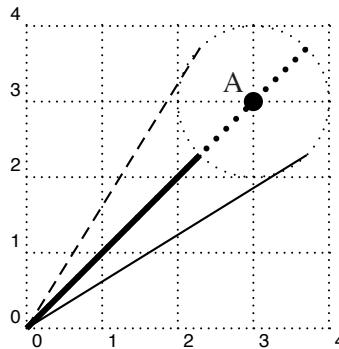


```

1 \SpecialCoor
2 \pnode(3,3){A}\psdot[dotscale=2](A)\uput[45](A){A}
3 \psline([nodesep=1]A)
4 \psline[linestyle=dashed]([nodesep=-1]A)
5 \psline[linestyle=dotted,linewidth=0.08]([offset=1]A)
6 \psline[linewidth=0.08]([nodesep=-1,offset=-1]A)

```

³⁹The `rand` PostScript operator generate an integer random number between 0 and $2^{31}-1$, `501 mod` allow to restrict it between 0 and 500 and `100 div` to transform it in a real number between 0 and 5. Then we use the `sethsbcolor` operator to fix the color of the dot, fixing randomly between 0 and 1 the Hue composante value (see [51] for more details.)

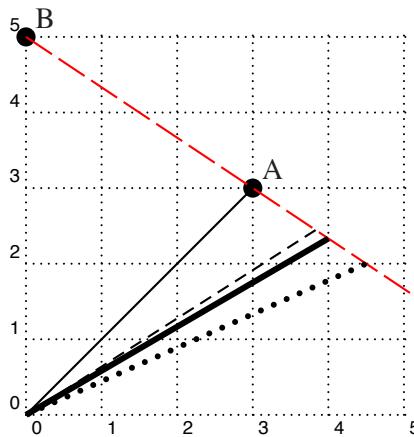


```

1 \SpecialCoor
2 \pnode(3,3){A}\psdot[dotscale=2](A)\uput[135](A){A}
3 \pscircle[linestyle=dotted](A){1}
4 \psline([nodesep=1,angle=-45]A)
5 \psline[linestyle=dashed]([nodesep=-1,angle=-45]A)
6 \psline[linestyle=dotted,linewidth=0.08]([offset=1,angle=-45]A)
7 \psline[linewidth=0.08]([offset=1,angle=135]A)

```

([par]{nodeB}nodeA) The position relative to *nodeA*, on the virtual line joining *nodeA* and *nodeB*, determined using the **angle**, **nodesep**, **Xnodesep**, **Ynodesep** and **offset** optional parameters. **Xnodesep** and **Ynodesep** allows to specify vertical and horizontal increments.



```

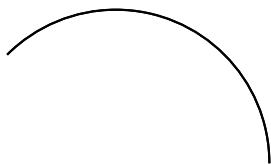
1 \SpecialCoor
2 \pnode(3,3){A}\psdot[dotscale=2](A)\uput[45](A){A}
3 \pnode(0,5){B}\psdot[dotscale=2](B)\uput[45](B){B}
4 \psline[linestyle=dashed,dash=0.4 0.1,linecolor=red]
5 (B)([nodesep=-2.5]{B}A)
6 \psline(A)
7 \psline[linestyle=dashed]([nodesep=-1]{B}A)
8 \psline[linewidth=0.08]([Ynodesep=-1]{B}A)
9 \psline[linestyle=dotted,linewidth=0.08]([Xnodesep=-1]{B}A)

```

\SpecialCoor also lets you specify angles in several ways:

num A number, as usual, with units given by the **\degrees** command.

(coor) A coordinate, indicating where the angle points to. Be sure to include the (), in addition to whatever other delimiters the angle argument uses. For example, the following are two ways to draw an arc of .8 inch radius from 0 to 135 degrees:



```

1 \SpecialCoor
2 \psarc(0,0){.8in}{0}{135}
3 \psarc(0,0){.8in}{0}{(-1,1)}

```

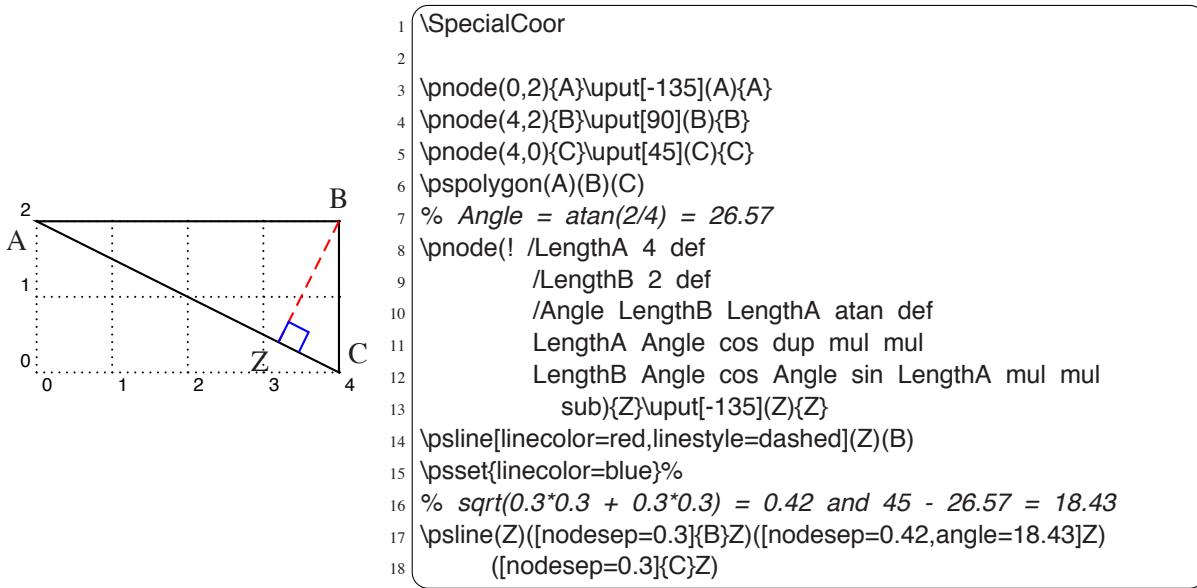
!ps Raw PostScript code. *ps* should expand to a number. The same units are used as with *num*.

The command

\NormalCoor

disables the `\SpecialCoor` features.

Here is an example for geometric drawing, which use some PostScript computations and node references.⁴⁰



55 Coils and zigzags



The file `pst-coil.tex/pst-coil.sty` (along with the header file `pst-coil.pro`) defines the following graphics objects for coils and zigzags:

```

\pscoil*[par]{arrows}(x0,y0)(x1,y1)
\psCoil*[par]{angle1}{angle2}
\pszigzag*[par]{arrows}(x0,y0)(x1,y1)

```

These graphics objects use the following parameters:

<code>coilwidth=dim</code>	Default: 1cm
<code>coilheight=num</code>	Default: 1
<code>coilarm=dim</code>	Default: .5cm
<code>coilaspect=angle</code>	Default: 45
<code>coilinc=angle</code>	Default: 10

All coil and zigzag objects draw a coil or zigzag whose width (diameter) is `coilwidth`, and with the distance along the axes for each period (360 degrees) equal to

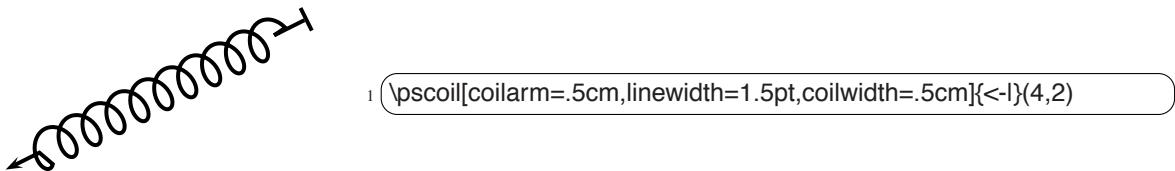
⁴⁰For such Euclidian geometric drawings, the ‘pst-eucl’ contribution package from Dominique Rodriguez [59] offer many high level and powerful functions.

coilheight x coilwidth.

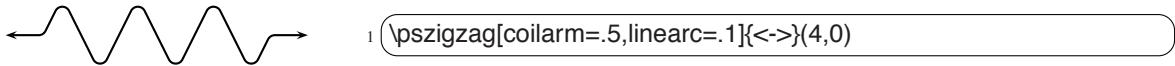
Both **\pscoil** and **\psCoil** draw a “3D” coil, projected onto the xz-axes. The center of the 3D coil lies on the yz-plane at angle **coilaspect** to the z-axis. The coil is drawn with PostScript’s **lineto**, joining points that lie at angle **coilinc** from each other along the coil. Hence, increasing **coilinc** makes the curve smoother but the printing slower. **\pszigzag** does not use the **coilaspect** and **coilinc** parameters.

\pscoil and **\pszigzag** connect (x_0, y_0) and (x_1, y_1) , starting and ending with straight line segments of length **coilarmA** and **coilarmB**, resp. Setting **coilarm** is the same as setting **coilarmA** and **coilarmB**.

Here is an example of **\pscoil**:

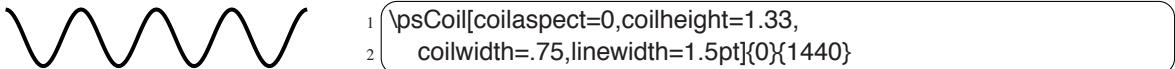


Here is an example of **\pszigzag**:



Note that **\pszigzag** uses the **linearc** parameters, and that the beginning and ending segments may be longer than **coilarm** to take up slack.

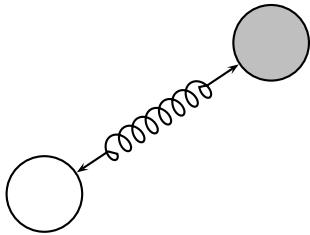
\psCoil just draws the coil horizontally from *angle1* to *angle2*. Use **\rput** to rotate and translate the coil, if desired. **\psCoil** does not use the **coilarm** parameter. For example, with **coilaspect=0** we get a sine curve:



`pst-coil.tex` also contains coil and zigzag node connections. You must also load `pst-node.tex/pst-node.sty` to use these. The node connections are:

```
\ncccoil*[par]{arrows}{nodeA}{nodeB}
\nczigzag*[par]{arrows}{nodeA}{nodeB}
\pccoil*[par]{arrows}{(x1,y1)(x2,y2)}
\pczigzag*[par]{arrows}{(x1,y1)(x2,y2)}
```

The end points are chosen the same as for **\ncline** and **\pcline**, and otherwise these commands work like **\pscoil** and **\pszigzag**. For example:



56 Overlays

Overlays are mainly of interest for making slides, and the overlay macros described in this section are mainly of interest to TeX macro writers who want to implement overlays in a slide macro package. For example, Seminar [78], a L^AT_EX document class for slides and notes, uses PSTricks to implement overlays.

Overlays are made by creating an `\hbox` and then outputting the box several times, printing different material in the box each time. The box is created by the commands⁴¹

```
\overlaysbox
stuff
\endoverlaysbox
```

The material for overlay *string* should go within the scope of the command

```
\psoverlay{string}
```

string can be any string, after expansion. Anything not in the scope of any `\psoverlay` command goes on overlay `main`, and material within the scope of `\psoverlayall` goes on all the overlays. `\psoverlay` commands can be nested and can be used in math mode.

The command

```
\putoverlaysbox{string}
```

then prints overlay *string*.

Here is an example:

```
1 \overlaysbox
2 \psoverlay{all}
3 \psframebox[framearc=.15,linewidth=1.5pt]{%
4 \psoverlay{main}}
```

⁴¹L^AT_EX users can instead write:

`\begin{overlaysbox} stuff \end{overlaysbox}`

```

5 \parbox{3.5cm}{\raggedright
6 Foam Cups Damage Environment {\psoverlay{one} Less
7 than Paper Cups,} Study Says.}}
8 \endoverlays
9
10 \putoverlays{main}{hspace{.5in}}\putoverlays{one}

```

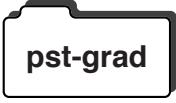
Foam Cups Damage
Environment

Study Says.

Less
than Paper Cups,

Driver notes: Overlays use `\pstVerb` and `\pstverbsscale`.

57 The gradient fill style



The file `pst-grad.tex/pst-grad.sty`, along with the PostScript header file `pst-grad.pro`, defines the gradient **fillstyle**, for graduated shading.⁴² This **fillstyle** uses the following parameters:

gradbegin=color **Default: 0 .1 .95**

The starting and ending color (default color is a dark blue).

gradend=color **Default: 0 1 1**

The color at the midpoint (default color is cyan).

gradlines=int **Default: 500**

The number of lines. More lines means finer gradation, but slower printing.

gradmidpoint=num **Default: .9**

The position of the midpoint, as a fraction of the distance from top to bottom. *num* should be between 0 and 1.

gradangle=angle **Default: 0**

The image is rotated by *angle*.

⁴²The ‘`pst-slpe`’ contribution package from Martin Giese [64] offer extended functionalities, like radial and concentric gradients and multicolored ones (for rainbow effects.)

gradbegin and **gradend** should preferably be `rgb` colors, but `gray` and `cmyk` colors should also work. The definitions of the colors `gradbegin` and `gradend` are:

```
1 gradbegin = 0 0.1 0.95  
2 gradend   = 0 1   1
```

Here are two ways to change the gradient colors:

```
1 \psset{gradbegin=blue}
```

or, with PLAIN \TeX and \LaTeX (using the ‘color’ package):

```
1 \definecolor{rgb}{gradbegin}{1,0.4,0}
```

and with Con \TeX t:

```
1 \definecolor{gradbegin}{r=1,g=0.4,b=0}
```

Try these examples:



```
1 \psframe[fillstyle=gradient,gradmidpoint=1,gradangle=45](4,2)
```



```
1 \psellipse[linestyle=none,fillstyle=gradient,gradmidpoint=0.5,  
2   gradangle=90,gradbegin=Pink,gradend=red](2,1)(2,1)
```

58 Typesetting text along a path



The file `pst-text.tex/pst-text.sty` defines the command `\pstextpath`, for typesetting text along a path. It is a remarkable trick, but there are some caveats:

- ‘pst-text’ only works with certain DVI-to-PS drivers. Here is what is currently known:
 - It works with Rokicki’s `dvips`.
 - “Does not work” means that it has no effect, for better or for worse.

- This may work with other drivers. The requirement is that the driver only use PostScript’s `show` operator, unbound and unloaded, to show characters.
- You must also have installed the PostScript header file `pst-text.pro`, and `\psheader` must be properly defined in `pstricks.con` for your driver.
- Like other PSTricks that involve rotating text, this works best with PostScript (outline) fonts.
- PostScript rendering with ‘`pst-text`’ is slow.

Here is the command:

`\pstextpath[pos](x,y){graphics object}{text}`

text is placed along the path, from beginning to end, defined by the PSTricks graphics object. (This object otherwise behaves normally. If you don’t want it to appear, set `linestyle=none`.)

text can only contain characters. No TeX rules, no PSTricks, and no other `\special`’s. (These things don’t cause errors; they just don’t work right.) Math mode is OK, but math operators that are built from several characters (e.g., large integral signs) may break. Entire boxes (e.g., `\parbox`) are OK too, but this is mainly for amusement.

l justify on beginning of path

pos is either c center on path

r justify on end of path.

The default is l.

(x, y) is an offset. Characters are shifted by distance *x* along path, and are shifted up by *y*. “Up” means with respect to the path, at whatever point on the path corresponding to the middle of the character. (x, y) must be Cartesian coordinates. Both coordinates use `\psunit` as the default. The default coordinate is $(0, \TPoffset)$, where `\TPoffset` a command whose default value is $-.7\text{ex}$. This value leads to good spacing of the characters. Remember that *ex* units are for the font in effect when `\pstextpath` occurs, not inside the *text* argument.

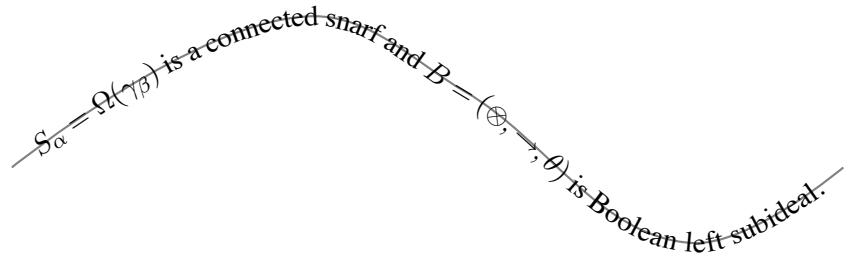
More things you might want to know:

- Like with `\rput` and the graphics objects, it is up to you to leave space for `\pstextpath`.
- Results are unpredictable if *text* is wider than length of path.
- `\pstextpath` leaves the typesetting to TeX. It just intercepts the `show` operator to remap the coordinate system.

```

1 \pstextpath[c]%
2 {\pscurve[linecolor=gray](0,1)(4,3)(6,2)(9,0)(11,1)}%
3 { $S_\alpha = \Omega(\beta)$  is a connected snarf and}
4 { $B = (\oplus, \rightarrow, \theta)$  is Boolean left subideal.}

```



Centre National de la
Recherche Scientifique

In principle, it is possible to use parbox, but let's see what really happens. It seems hard to believe that someone would want to do this.

```

1 \LARGE
2 \psset{linestyle=none}%
3 \pstextpath[c]{\psarcn(0,0){1.8}{180}{0}}{Centre National de la}
4 \pstextpath[c]{\psarc(0,0){1.8}{180}{0}}{Recherche Scientifique}

```

```

1 \pstextpath[c]{(0,0)}{\psarcn[linestyle=none]{0,-6}{4}{180}{0}}%
2 {\parbox{4cm}}%
3 In principle, it is possible to use parbox,
4 but let's see what really happens. It seems hard
5 to believe that someone would want to do this.}}

```

And, last, a poem of the *calligramme* form, *La colombe poignardée et le jet d'eau*, by the french poet Guillaume Appolinaire [3], re-typeset from the original⁴³ and first published in *Les Cahiers GUTenberg* in 1994 [22].

```

1 \small
2 \psset{linestyle=none}%
3 \pstextpath[r]{\pscurve(1.5,8)(3,9)(4.45,6.5)}%
4 {Tous les souvenirs de naguere}
5 \pstextpath[r]{\pscurve(1,7)(2,7.5)(4.45,6)}%
6 {O mes amis partis en guerre}
7 \pstextpath[r]{\pscurve(1,6)(2,6.7)(4.45,5.2)}%
8 {Jaillissent vers le firmament}
9 %

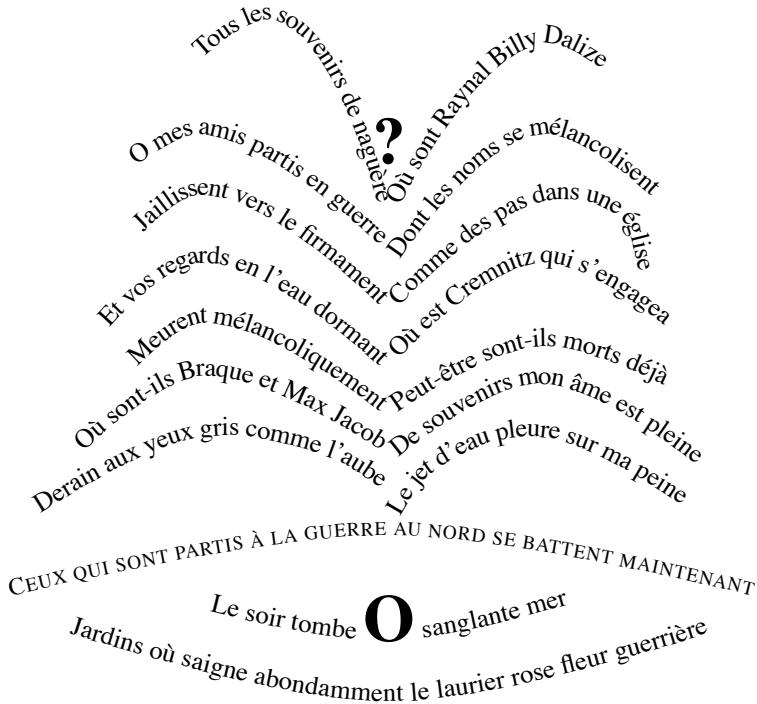
```

⁴³Only the code for few verses is shown here, but the complete code will be found in the source file of this documentation.

```

10 \pstextpath[c]{\pscurve(2.1,1.2)(4.5,0.8)(6.9,1.3)}
11 {Le soir tombe \textbf{Huge O} sanglante mer}
12 \pstextpath[c]{\pscurve(0,1)(4.5,0)(9,1)}
13 {Jardins o^f9 saigne abondamment le laurier rose fleur guerri^e8re}

```



59 Stroking and filling character paths

pst-char

The file `pst-char.tex/pst-char.sty` defines the command:

\pscharpath*[par]{text}

It strokes and fills the *text* character paths using the PStricks **linestyle** and **fillstyle**.

The restrictions on DVI-to-PS drivers listed on page 152 for `\pstextpath` apply to `\pscharpath`. Furthermore, only outline (PostScript) fonts are affected.

```

1 \DeclareFixedFont{\Sf}{T1}{phv}{b}{n}{3.5cm}
2 \DeclareFixedFont{\Rm}{T1}{ptm}{m}{n}{3mm}
3
4 \pscharpath[linestyle=none,fillstyle=gradient,gradmidpoint=0.5,
5           gradbegin=PaleGreen,gradend=ForestGreen]{\Sf TeX}

```

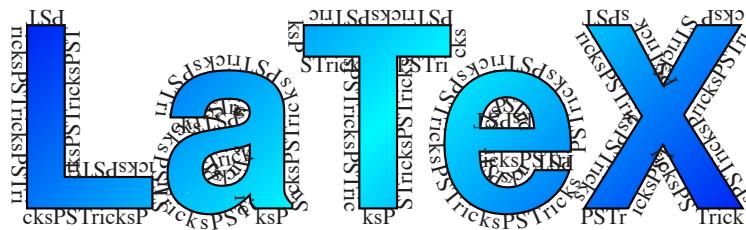
TeX

With the optional *, the character path is not removed from the PostScript environment at the end. This is mainly for special hacks. For example, you can use **\pscharpath*** in the first argument of **\pstextpath**, and thus typeset text along the character path of some other text. However, you cannot combine **\pscharpath** and **\pstextpath** in any other way. E.g., you cannot typeset character outlines along a path, and then fill and stroke the outlines with **\pscharpath**.

```

1 \pstextpath(0,-2mm){\pscharpath*[fillstyle=gradient,
2                                     gradangle=45,gradmidpoint=0.5]{\Sf LaTeX}}%
3 {\Rm\multido{\i=1+1}{60}{\PSTricks}}

```



The command⁴⁴

\pscharclip*[par]{text} ... \endpscharclip

works just like **\pscharpath**, but it also sets the clipping path to the character path. You may want to position this clipping path using **\rput** inside **\pscharclip**'s argument. Like **\psclip** and **\endpsclip**, **\pscharclip** and **\endpscharclip** should come on the same page and should be properly nested with respect to TeX groups (unless **\AltClipMode** is in effect).⁴⁵

```

1 \begin{pscharclip}[linewidth=3pt]{\rput[t](0,0){\Sf ConTeXt}}
2   \rput{58}{%
3     \color{red}
4     \begin{minipage}{11cm}
5       \multido{\i=1+1}{220}{\PSTricks }
6     \end{minipage}}
7 \end{pscharclip}

```

⁴⁴LaTeX users can instead write:

\begin{pscharclip}*{par}{text} ... \end{pscharclip}

⁴⁵Another way to do this is to use the **\pscharpath** macro with the **\psboxfill** one (see Section 50.3.)

ConTeXt

60 Importing EPS files

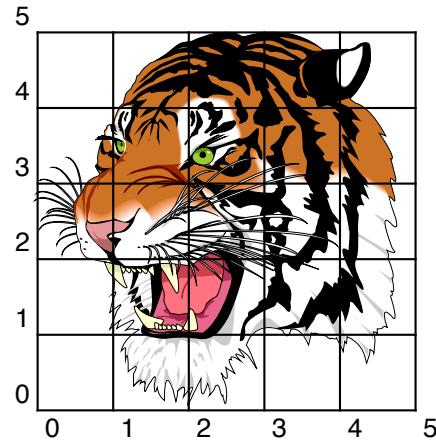
PSTricks does not come with any facility for including Encapsulated Post-Script files, because there are other very good and well-tested macros for exactly that, specially the `graphicx` package for L^AT_EX, usable also with T_EX.

What PSTricks *is* good for is embellishing your EPS picture. You can include an EPS file in the argument of `\rput`, as in

```
1 \rput(3,3){\includegraphics{myfile}}
```

and hence you can include an EPS file in the `\pspicture` environment. Turn on `\psgrid`, and you can find the coordinates for whatever graphics or text you want to add. This works even when the picture has a weird bounding box, because with the arguments to `\pspicture` you control the bounding box from T_EX's point of view.⁴⁶

```
1 \begin{pspicture}(5,5)
2   \rput[b](0,0){\includegraphics[scale=0.25]{tiger}}
3   \psgrid[subgriddiv=0]
4 \end{pspicture}
```



If you do not want to adjust the dimension of the `pspicture` environments by tries and errors, you can store the image in a T_EX box (`\pst@boxg` is

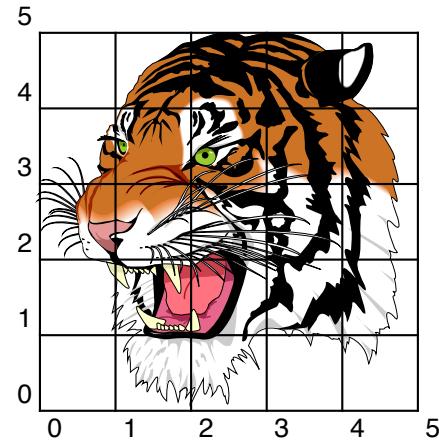
⁴⁶Of course, with ConTeXt, use the PLAIN T_EX syntax for the `\pspicture` environment and the macro `\externalfigure`.

the name of an internal temporary PSTricks box), and ask for its width and height:

```

1 \savebox{\pst@boxg}{\includegraphics[scale=0.25]{tiger}}
2 \begin{pspicture}(\wd\pst@boxg,\ht\pst@boxg)
3   \rput[bl](0,0){\usebox{\pst@boxg}}
4   \psgrid[subgriddiv=0]
5 \end{pspicture}

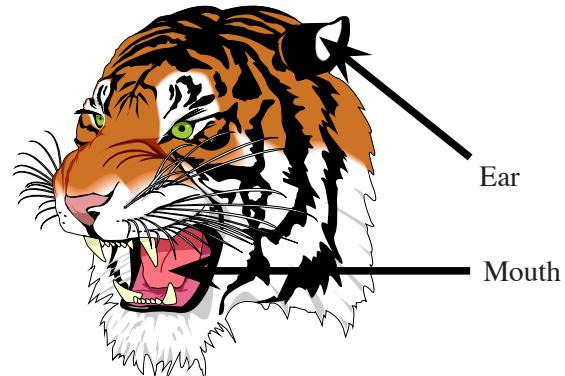
```



```

1 \begin{pspicture}(5,5)
2   \rput[bl](0,0){\includegraphics[scale=0.25]{tiger}}
3   \psset{linewidth=0.1,arrows=->,arrowscale=2}%
4   \psline(6,3)(4,4.6)\uput[-45](6,3){Ear}
5   \psline(6,1.5)(2,1.5)\uput[0](6,1.5){Mouth}
6 \end{pspicture}

```



This isn't always the best way to work with an EPS file, however. If the PostScript file's bounding box is the size you want the resulting picture to be, after your additions, then try

```

1 \hbox{picture objects \includegraphics{file}}

```

This will put all your picture objects at the lower left corner of the EPS file.

If you need to determine the bounding box of an EPS file, then you can try of the automatic bounding box calculating programs, such as `bbfig` (distributed with Rokicki's `dvips`). However, all such programs are easily fooled; the only sure way to determine the bounding box is visually. `\psgrid` is a good tool for this.

61 Exporting EPS files



You must load `pst-eps.tex/pst-eps.sty` to use the PStricks macros described in this section.

Exporting a PStricks graphic as an EPS file is specially useful in some circumstances (to put such graphic on a Web page, after conversion of the EPS file in JPEG or PNG format, to generate animated graphics (see the Section B), etc.) If you want to export an EPS file that contains both graphics and text, you must use the `\TeXtoEPS` environment and a DVI-to-PS driver that supports such a feature. If you just want to export pure graphics, then you can use the `\PSTtoEPS` command. Both of these options are described in this section.

Rokicki's `dvips` support an `-E` option for creating EPS files from `\TeX` .dvi files. E.g.,

```
dvips foo.dvi -E -o foo.eps
```

Your document should be a single page (do not forget to suppress the header, footer, page number, etc. to do not include extra material in the resulting file!) `dvips` will find a tight bounding box that just encloses the printed characters on the page. This works best with outline (PostScript) fonts, so that the EPS file is scalable and resolution independent.

There are two inconvenient aspects of this method. You may want a different bounding box than the one calculated by `dvips` (in particular, `dvips` ignores all the PostScript generated by PStricks when calculating the bounding box), and you may have to go out of your way to turn off any headers and footers that would be added by output routines.

PStricks contains an environment that tries to get around these two problems⁴⁷:

```
\TeXtoEPS  
stuff  
\endTeXtoEPS
```

⁴⁷`\TeX` users can instead write:

```
\begin{TeXtoEPS} stuff \end{TeXtoEPS}
```

This is all that should appear in your document, but headers and whatever that would normally be added by output routines are ignored. `dvips` will again try to find a tight bounding box, but it will treat `stuff` as if there was a frame around it. Thus, the bounding box will be sure to include `stuff`, but might be larger if there is output outside the boundaries of this box. If the bounding box still isn't right, then you will have to edit the

```
1 %%BoundingBox lly urx ury
```

specification in the EPS file by hand.

If your goal is to make an EPS file for inclusion in other documents, then `dvips -E` is the way to go. However, it can also be useful to generate an EPS file from PSTricks graphics objects and include it in the same document,⁴⁸ rather than just including the PSTricks graphics directly, because `TeX` gets involved with processing the PSTricks graphics only when the EPS file is initially created or updated. Hence, you can edit your file and preview the graphics, without having to process all the PSTricks graphics each time you correct a typo. This speed-up can be significant with complex graphics such as `\listplot`'s with a lot of data.

To create an EPS file from PSTricks graphics objects, use

\PSTtoEPS[*par*]{*file*}{graphics objects**}**

The file is created immediately, and hence you can include it in the same document (after the `\PSTtoEPS` command) and as many times as you want. Unlike with `dvips -E`, only pure graphics objects are processed (e.g., `\rput` commands have no effect).

`\PSTtoEPS` cannot calculate the bounding box of the EPS file. You have to specify it yourself, by setting the following parameters:

bbllx= <i>dim</i>	Default: Opt
bblly= <i>dim</i>	Default: Opt
bburx= <i>dim</i>	Default: Opt
bbury= <i>dim</i>	Default: Opt

Note that if the EPS file is only to be included in a PSTricks picture with `\rput` you might as well leave the default bounding box.

`\PSTricksEPS` also uses the following parameters:

makeeps= <i>none/new/all/all*</i>	Default: new
--	---------------------

This parameter determines which `\PSTtoEPS` commands just skip over their arguments, and which create files, as follows:

⁴⁸See the preceding section on importing EPS files.

none No files are created.

new Only those files not found *anywhere on the system* are created
(take care that you must remove the files to re-create them!)

all All files are created.

all* All files are created, but you are asked for approval before existing files are deleted.

headerfile=files

Default:

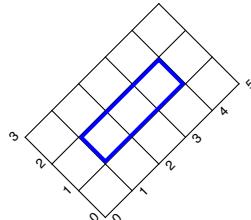
This parameter is for specifying PostScript header files that are to be included in the EPS file. The argument should contain one or more file names, separated by commas. If you have more than one file, however, the entire list must be enclosed in braces {}.

headers=none/all/user

Default: none

When **none**, no header files are included. When **all**, the header files used by PSTricks plus the header files specified by the **headerfile** parameter are included. When **user**, only the header files specified by the **headerfile** parameter are included. If the EPS file is to be included in a TeX document that uses the same PSTricks macros and hence loads the relevant PSTricks header files anyway (in particular, if the EPS file is to be included in the same document), then **headers** should be **none** or **user**.

You can either store the graphic in an unique EPS file:



```
1 \PSToEPS[bbllx=-0.2,bbll=-0.2,bburx=5,bbury=3]{Frame.eps}%
2 \psgrid[subgriddiv=0](5,3)
3 \psframe[linecolor=blue,linewidth=0.1](1,1)(4,2)
4
5 \includegraphics[angle=45,scale=0.5]{Frame}
```

or put the content in a personal *header* file, using the **\psNewHeader** and **\psDefineProc** macros in the preamble of the file, and reusing it later (in the same source file or, of course, in another TeX file), using the **\psUseHeader** and **\psUseProc** macros.

\psNewHeader{header file}{content}

\psDefineProc{procedure name}{graphics objects}

\psUseHeader{header file}

\psUseProc[par]{procedure name}

There are four special parameters to specify how to handle the graphic when it is reused:

GraphicsRef={x0,y0}

Default: none

The reference point to use (the coordinates must be given between braces.) These values will be set by the **origin** parameter.

Translation={x1,y1}

Default: none

The point where the graphic will be translated (the coordinates must be given between braces.)

Rotation=num

Default: none

The angle to use.

Scale=num1 num2

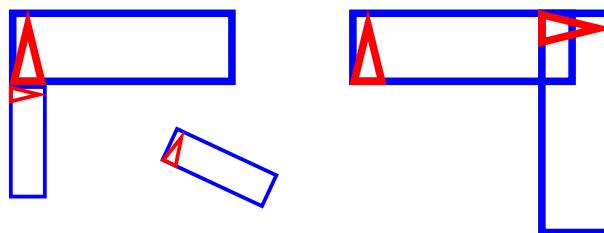
Default: none

The scaling factors to use (if only one number is given, it will be used both for horizontal and vertical scalings.)

```

1 \psNewHeader{Frame.pro}%
2  \psDefineProc{Frame}{%
3   \psset{linewidth=0.1}%
4   \psframe[linecolor=blue](3,1)
5   \pstriangle[linecolor=red](0.25,0)(0.5,1)}}
6
7 \psUseHeader{Frame.pro}
8
9 % ...
10
11 %\begin{document}
12 %
13 %
14
15 \psUseProc{Frame}
16 \psUseProc[Rotation=-90,Scale=0.5]{Frame}
17 \psUseProc[Translation={4.5,0}]{Frame}
18 \psUseProc[Translation={2,-1},Rotation=-25,Scale=0.5]{Frame}
19 \psUseProc[Translation={7,1},Rotation=-90]{Frame}

```



XII

Writing high-level macros⁴⁹

This chapter is intended only for *advanced* users who want to develop new high-level macros or new specialized packages. It is a detailed tutorial which progressively introduces many techniques which are commonly used in PStricks packages.

62 Overview



The file `pst-key.tex/pst-key.sty`, written by David Carlisle in 1997, defines an interface between PStricks and the ‘keyval’ package [35], written too by David Carlisle, which implements a general mechanism to flexibly manage the parameters of a macro. It uses the `key=value` syntax and is a generic implementation of the original internal mechanism which PStricks uses to handle parameters.

This package can be used with PLAIN T_EX, L^AT_EX and ConT_EXt. Developers are strongly encouraged to create packages that are *generic*, not limited to L^AT_EX or another T_EX flavor. Use of the ‘pst-key’ package will make it easier for this to be done. Of course, other precautions must also be taken to ensure that packages are generic.

The ‘keyval’ prefix used by ‘pst-key’ is `psset`, so the macro

```
1 \setkeys{psset}{param1=value1,...,paramN=valueN}
```

will initialize the defined parameters, inside the T_EX group where it is called.

To manage parameters globally or locally to a macro, the general method is the following (sometimes, the grouping mechanism must be done otherwise – see for instance the `\PstDotsScaled` macro in Section 63.5):

```
1 \setkeys{psset}{...}% Assignment of global parameters
2
3 \def\PstXxxx{\def\pst@par{}\pst@object{PstXxxx}}
4
5 \def\PstXxxx@{{\% Two braces, because parameter
```

⁴⁹This chapter was written in 2003 by Denis Girou.

```

6          % assignments must be local
7 \use@par% Assignment of local parameters
8 % The code itself, which can of course include calls
9 % to auxiliary macros
10 % ...
11 }}

```

The `\pst@object` macro will assign the optional parameter to the `\pst@par` macro. The code above is essentially the same as the following code, which lacks only management of the * convention:

```

1 \setkeys{psset}{...}% Assignment of global parameters
2
3 \def\PstXxxx{%
4 % Test for optional parameters
5 \@ifnextchar[{\PstXxxx@i}{\PstXxxx@i[]}}
6
7 \def\PstXxxx@i[#1]{% Two braces, because parameter
8 % assignments must be local
9 \setkeys{psset}{#1}% Assignment of local parameters
10 % ...
11 }

```

Take special care with naming conventions and try to choose a consistent naming scheme! As in the next examples, a good programming practice is to start the names of all the macros and parameters relative to the same subject with a common prefix. Additionally, it is good practice to use the @ character in the names of macros which are not intended for direct access by users.

63 Usage

Most often, we need to define *integer*, *real*, *string*, *boolean*, *length* or *coordinate* optional parameters. *T_EX* registers are a scarce resource. Care should always be taken to avoid using a register to store a value if it can be stored in a macro instead. Integer, real, string, and length parameters, aside from some exceptional cases, can be stored in macros. When computations must be done on integer or dimension parameters which are stored in macros, the macros can be evaluated and the values assigned to temporary count or dimension registers. PSTricks provides six temporary count registers (`\pst@cpta`, ..., `\pst@cnth`) and six temporary dimension registers (`\pst@dpta`, ..., `\pst@dimh`) for this purpose. Computations can then be carried out using the temporary registers. Since PSTricks also uses these temporary registers to carry out some of its internal computations, care

must be taken to avoid using temporary registers whose values might be altered the particular PSTricks macros which are used. There are many examples in the code which follows in this section and in the PSTricks code itself of the use of macros for storing integer and real values and temporary registers for manipulating these values.

63.1 Integer parameters

For *integer* parameters, use the `\pst@getint` macro to assign the value to a personal macro. Do not define new TeX counters, which are a scarce resource. See the examples below and the PSTricks code itself for more details.

```
1 \define@key{psset}{Integer}{\pst@getint{\#1}\PstXxx@Integer}
```

Here is an example which prints a partial multiplication table.

```
1 % "Start", "End" and "Value" parameters
2 \define@key{psset}{Start}{\pst@getint{\#1}\MultTable@Start}
3 \define@key{psset}{End}{\pst@getint{\#1}\MultTable@End}
4 \define@key{psset}{Value}{\pst@getint{\#1}\MultTable@Value}
5 \setkeys{psset}{Start=1,End=5,Value=2}% Default values
6
7 % Main macro for "multiplication table" object
8 \def\MultTable{\def\pst@par{}\pst@object{MultTable}}
9
10 \def\MultTable@#1{%
11   \use@par% Assignment of local parameters
12   \pst@cnth=\MultTable@End
13   \advance\pst@cnth-\MultTable@Start
14   \advance\pst@cnth\@ne
15   \multido{\i=Value=\MultTable@Start+\@ne}{\pst@cnth}{%
16     \pst@cntg=\i
17     \multiply\pst@cntg\MultTable@Value
18     \iValue$times$\MultTable@Value = \the\pst@cntg
19     \ifnum\multidocount=\pst@cnth\else; \fi}}
20
21 \MultTable
22
23 \MultTable[Value=9]
24
25 \setkeys{psset}{Value=6}
26
27 \MultTable[Start=6,End=11]
28
29 \MultTable[Start=19742,End=19742]
```

$$1 \times 2 = 2; 2 \times 2 = 4; 3 \times 2 = 6; 4 \times 2 = 8; 5 \times 2 = 10$$

$$1 \times 9 = 9; 2 \times 9 = 18; 3 \times 9 = 27; 4 \times 9 = 36; 5 \times 9 = 45$$

$$6 \times 6 = 36; 7 \times 6 = 42; 8 \times 6 = 48; 9 \times 6 = 54; 10 \times 6 = 60; 11 \times 6 = 66$$

$$19742 \times 6 = 118452$$

We can also handle a special behavior with the `*` convention. Here, the result of the multiplication is put in a colored circle.

```

1 \def\MultTable@#1{%
2 \use@par% Assignment of local parameters
3 \pst@cnth=\MultTable@End
4 \advance\pst@cnth-\MultTable@Start
5 \advance\pst@cnth\@ne
6 \multido{\i=Value=\MultTable@Start+\@ne}{\pst@cnth}{%
7   \pst@cntg=\i
8   \multiply\pst@cntg\MultTable@Value
9   \i\Value$ \times \$ \MultTable@Value =
10  \if@star
11    {\pscirclebox*[fillstyle=solid,fillcolor=red,framesep=0.02]{%
12      \textcolor{yellow}{\the\pst@cntg}}}{%
13  \else
14    \the\pst@cntg
15  \fi
16  \ifnum\multidocount=\pst@cnth\else; \fi}}
17 \MultTable[Value=13,Start=7,End=10]
18 \MultTable*[Value=13,Start=7,End=10]
19
20

```

$$7 \times 13 = 91; 8 \times 13 = 104; 9 \times 13 = 117; 10 \times 13 = 130$$

$$7 \times 13 = \textcolor{red}{91}; 8 \times 13 = \textcolor{red}{104}; 9 \times 13 = \textcolor{red}{117}; 10 \times 13 = \textcolor{red}{130}$$

63.2 Real parameters

real parameters (used to define scales, ratios, etc.) and *dimension* parameters should be stored in macros in order to avoid consuming scarce resources. See the examples below and the PSTricks code itself for more details.

So, define a real value by:

```

1 \define@key{psset}{Real}{\pst@checknum{#1}\PstXxxx@Real}}

```

if this will be “truly” a real, as the `\pst@checknum` macro allow in supplement to check that the number is correctly formatted.

Lengths are a special case. They can be real values, in which case the current unit will be used, or they can be dimensions, with a unit specified. The **\pst@getlength** macro allow to scale the “real” values with the current unit. The result will be given in T_EX points, without the pt unit, so take care to add it (it is better to use the **\p@** macro for that) each time you will use this variable.

```

1 \define@key{psset}{Length}{%
2   \pst@getlength{#1}{\PstXxxx@Length}}}
```

Nevertheless, take care to the pitfall that with **\pst@getlength** the values are scaled at assignation. The line **\PstXxxx[unit=0.2,Length=1]** will use a length of 2mm (if the current unit was of 1cm), as expected. But the construction

```

1 \setkeys{psset}{Length=3}
2 ...
3 \PstXxxx[unit=0.2]
```

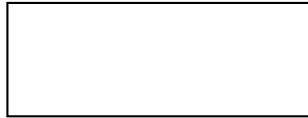
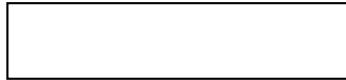
will use a length of 1cm, as it assignment has been done before. So, if such order of assignment is expected, it is better to define the length parameter as a real value, and to use internally the **\pssetlength** macro. It does the same thing than **\pst@getlength**, but on *dimensions*, not *macros*. And the **\psaddtolength** macro allow to add a value to such dimension, scaled too by the current unit.

Angles are another special case. Angle parameters are real values, but they must be defined with the **\pst@getangle** macro so that “special” definitions (based on PostScript computations, for example, as in Section 54) can be used.

```

1 \define@key{psset}{Angle}{\pst@getangle{#1}{\PstXxxx@Angle}}
```

Here is an example which draws a bar and a wall one unit to the right, with the length of the bar being a parameter. It can either take an integer real or length value (we can verify on the picture that when we use an integer or real value, the current unit is taken in account, as for all PSTRicks dimensions. This is not true if we use a value explicitly associated to a unit length, as expected.) Note also the usage of the **\psaddtolength** macro to increase or decrease a length, which allows it to be scaled according to the current unit.



```

1 % "Length" parameter: length of the bar
2 \define@key{psset}{Length}{\edef\PstBar@Length{\#1}}
3 \setkeys{psset}{Length=3}% Default value
4
5 % Main macro for "bar" object
6 \def\PstBar{\def\pst@par{} \pst@object{PstBar}}
7
8 \def\PstBar@#1{%
9   \setkeys{psset}{framesep=1pt}%
10  \use@par% Assignment of local parameters
11  \psframe(\PstBar@Length,\@ne)
12  \pssetlength{\pst@dima}{\PstBar@Length}%
13  \psaddtolength{\pst@dima}{\@ne}%
14  \psline[linewidth=0.08]{\pst@dima,-0.5}{\pst@dima,1.5}
15  \pst@dimb=\pst@dima
16  \psaddtolength{\pst@dimb}{0.5}%
17  \psframe[linestyle=none,fillstyle=hlines]
18    (\pst@dima,-0.5){\pst@dimb,1.5}
19  \psaddtolength{\pst@dima}{0.25}%
20  \rput*[90]{\pst@dima,0.5}{Wall}}%
21
22 \rput(-2,3){\PstBar[Length=4.5]}
23 \rput(-2,0){\PstBar[Length=1.65cm]}
24 \rput(-2,-3){\PstBar[unit=1.5,Length=4cm]}

```

63.3 String parameters

String parameters should also be stored as personal macros using either:

```
1 \define@key{psset}{String}{\edef\PstXxxx@String{\#1}}
```

or

```
1 \define@key{psset}{String}{\def\PstXxxx@String{\#1}}
```

If the value of #1 can be evaluated correctly at the time of definition, the `\edef` alternative should be used, because it avoid side effects due to later expansion, as in the following case where the `\PstObject` called in the last line will have OLD for value of the `String` parameter if `\edef` is used, which is the original content at the time of the assignment of the parameter, rather than NEW which will be the result if `\def` is used.

```

1 \def\MyString{OLD}
2 \setkeys{psset}{String=\MyString}
3 ...
4 \def\MyString{NEW}
5 \PstObject

```

In some cases, however, expansion at the point of definition is not desired and the `\def` alternative must be used. One typical case is when a font specification is in the definition, such as `String=\Large text`, which cannot be evaluated at the point of definition.

If the string is used to define a specific macro, a powerful solution is to dynamically define the macro name according to the value of the string parameter given (for such an example, see how the `Form` parameter of the test tube in Section 64.2 is defined):

```

1 \define@key{psset}{String}{%
2 \def\PstXxxx@Macro{\@nameuse{PstXxxx@Macro@\#1}}}
3
4 \setkeys{psset}{String=BBBB}% Default value
5
6 % Macros for possible values
7 \def\PstXxxx@Macro@AAAA{...}
8 \def\PstXxxx@Macro@BBBB{...}
9 \def\PstXxxx@Macro@CCCC{...}
10
11 \def\PstXxxx{\def\pst@par{}\pst@object{PstXxxx}}
12
13 \def\PstXxxx@i{%
14   %
15   \expandafter\PstXxxx@Macro
16   %
17 }

```



```

1 % "Label" parameter: label to legend the support
2 \define@key{psset}{Label}{\def\PstBar@Label{\#1}}
3 \setkeys{psset}{Label=Wall}% Default value
4
5 \def\PstBar@i{%
6   %
7   \rput*(90){\pst@dima,0.5}{\PstBar@Label}}
8
9 \rput(-2,0){\PstBar[Label=\textcolor{red}{Support}]}

```

63.4 Boolean parameters

Boolean parameters should be defined using the `\newif` `\TeX` macro. Using the associated `\PstXxxx@Booleantrue` and `\PstXxxx@Booleanfalse` macros, the parameter can be set to true or false. The value inside square brackets below is the default setting of the parameter, set when the parameter is defined (so, `Boolean` would be equivalent to `Boolean=true`). This is optional, but obviously the usual convention will be to have `true` as the

default value, and not `false`, which would be the case if we do not explicitly define the default value to `true`.

```

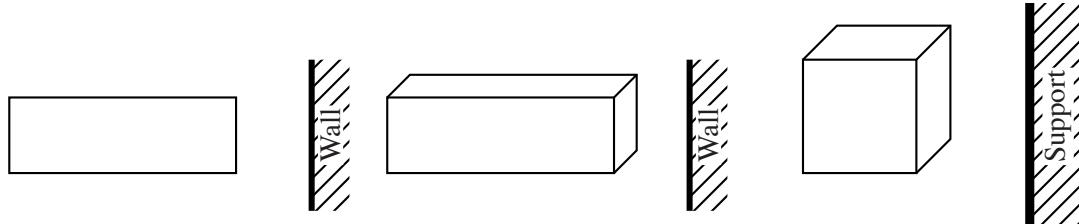
1 \newif\ifPstXxxx@Boolean
2 \define@key{psset}{Boolean}[true]{%
3   \nameuse{PstXxxx@Boolean#1}}

```

```

1 % "ThreeD" parameter: to emulate or not a 3d bar
2 \newif\ifPstBar@ThreeD
3 \define@key{psset}{ThreeD}[true]{%
4   \nameuse{PstBar@ThreeD#1}}
5
6 \setkeys{psset}{ThreeD=false}% Default value
7
8 \def\PstBar@i{%
9   \setkeys{psset}{framesep=1pt}%
10  \use@par% Assignment of local parameters
11  \psframe(\PstBar@Length,\@ne)
12  \pssetlength{\pst@dima}{\PstBar@Length}%
13  \ifPstBar@ThreeD
14    \pst@dimb=\pst@dima
15    \psaddtolength{\pst@dimb}{0.3}%
16    \psline(0,1)(0.3,1.3)(\pst@dimb,1.3)
17      (\pst@dimb,0.3)(\PstBar@Length,0)
18    \psline(\PstBar@Length,1)(\pst@dimb,1.3)
19  \fi
20  \psaddtolength{\pst@dima}{\@ne}%
21  \psline[linewidth=0.08](\pst@dima,-0.5)(\pst@dima,1.5)
22  \pst@dimb=\pst@dima
23  \psaddtolength{\pst@dimb}{0.5}%
24  \psframe[linestyle=none,fillstyle=hlines]
25    (\pst@dima,-0.5)(\pst@dimb,1.5)
26  \psaddtolength{\pst@dima}{0.25}%
27  \rput*{90}{\pst@dima,0.5}{\PstBar@Label}}}
28
29 \rput(-7.5,0){\PstBar}
30 \rput(-2.5,0){\PstBar[ThreeD=true]}
31 \rput(3,0){\PstBar[unit=1.5,Length=1,Label=Support,ThreeD=true]}

```



63.5 Coordinate arguments

A *coordinate* is a special kind of argument. We use the term *argument* rather than the term *parameter* because coordinates do not use the ‘pst-

`key`' package and are not usually handled by the methods described above. Nevertheless, as they are of common use, we will explain the main ways to manage them.

Two features are important: first, if possible, not only cartesian coordinates, but also all the *special* forms defined (see the Section 54) should be managed; and second, it is often necessary to manage unknown numbers of coordinates, of the $(x_0,y_0)(x_1,y_1)\dots(x_n,y_n)$ form.

We need to distinguish *low-level* (also called *basic*) objects using coordinates from *high-level* ones. You will probably not need to define new basic objects, as the existing ones cover the great majority of the foreseeing needs. Nevertheless, we will briefly explain here how to build them, using the examples (not very useful in practical life!) of *randomized* lines, polygons and dots.⁵⁰

Low-level objects

Four special kinds of macros are available to help build low-level objects, which allow management of *open* (two kinds of them, one with and one without optional arrows), *closed* and *special* objects. The four corresponding pair of macros are:

```

1 \begin{@OpenObj} ... \end{@OpenObj}
2 \begin{@AltOpenObj} ... \end{@OpenObj}
3 \begin{@ClosedObj} ... \end{@ClosedObj}
4 \begin{@SpecialObj} ... \end{@SpecialObj}
```

We will give examples for the first, third and fourth ones (the second is obviously very similar to the first one.)

First we define a real value parameter which is used to control the extent of the randomness effect:

```

1 % Random coefficient parameter
2 \define@key{psset}{RandomCoefficient}{%
3 \pst@checknum{#1}\PstRandomLine@RandomCoefficient}%
4
5 \setkeys{psset}{RandomCoefficient=1}% Default value
```

Then we define a macro `\PstRandomLine` which will be an *open* basic object, with possible arrows, that will be adapted from the `\psline` macro.

After dealing with the optional parameters, as usual, the `\PstRandomLine@i` macro is called. The management of the optional arrows is done by the

⁵⁰For complementary information, see in [79] the detailed explanation of how the `\pspolygon` macro is defined.

\pst@getarrows macro. Then the \begin@OpenObj macro initializes various things for this kind of object and the \pst@getcoors macro allows management of a sequence of coordinates of unknown length, which could simply be cartesian but also can be any kind of the *special* coordinates, accumulating their PostScript definitions in an internal variable (\pst@coors.)

The \PstRandomLine@ii macro stores the PostScript code which defines the Lineto PostScript operator which will join two points (in the \psline macro, it is simply the standard lineto operator, but if the **linear** parameter has been set, it will be a special form of an arc). Then the \tx@Line macro is inserted, which calls the Line PostScript operator, defined in the PSTricks header file pstricks.pro, which will do the *real* work of looping through the points and joining them. Finally, to finish the work, the macro \end@OpenObj is called, which carries out various tasks like managing the **border**, **doubleline** and **shadow** effects, visualizing the points themselves if the **showpoints** parameter has been set to true, etc. and writing the accumulated PostScript code.

The main task of the \PstRandomLine@iii macro is to define the Lineto PostScript operator which will be used to join the points. Here, we simply define it as lineto, but before applying this operator we add a random *noise* to both the horizontal and vertical coordinates (note that we will not do this for the first and last coordinates.) For this, we add to each component a value called **RandomValue**, computed by generating an integer random number between 1 and $2^{31}-1$, mapping it to the range 0 and 100 by applying the 101 mod modulo operator, then to a real number between 0 and 1 by using the 100 div operator, then to a real number between -0.5 and 0.5 by subtracting 0.5 by the 0.5 sub operator, then multiplying it by the current value of the **unit** parameter, then multiplying it by the **RandomCoefficient** value that we have defined.

```

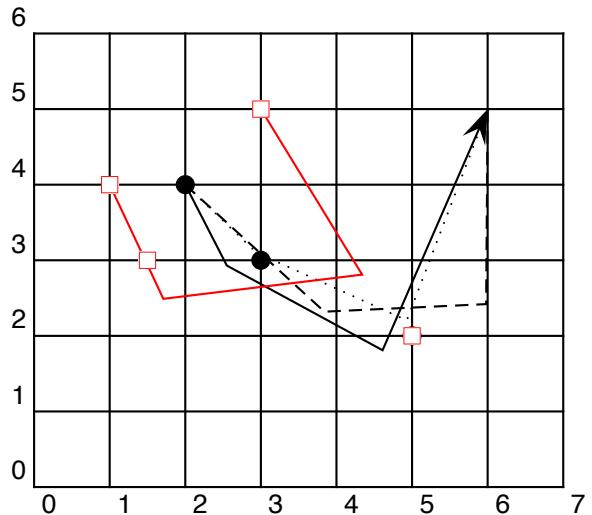
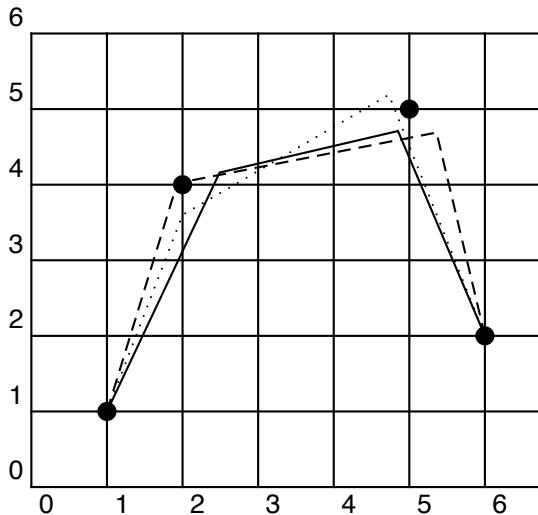
1 \def\PstRandomLine{\pst@object{PstRandomLine}}
2
3 \def\PstRandomLine@i{%
4 \pst@getarrows{%
5 \begin@OpenObj
6 \pst@getcoors[\PstRandomLine@ii]{}
7
8 \def\PstRandomLine@ii{%
9 \addto@pscode{\pst@cp \PstRandomLine@iii \tx@Line}{%
10 \end@OpenObj}}
11
12 \def\PstRandomLine@iii{%
13 /RandomValue {rand 100 mod 100 div 0.5 sub
14           \pst@number{\psunit} mul
15           \PstRandomLine@RandomCoefficient\space mul} def
16 /Lineto {exch RandomValue add exch RandomValue add lineto} def
17 \ifshowpoints true \else false \fi}
18
19 \pspicture(7,6)\psgrid[subgriddiv=0]

```

```

20 \PstRandomLine[showpoints=true,dotscale=2](1,1)(2,4)(5,5)(6,2)
21 \PstRandomLine[linestyle=dashed](1,1)(2,4)(5,5)(6,2)
22 \PstRandomLine[linestyle=dotted](1,1)(2,4)(5,5)(6,2)
23 \endpspicture
24 \hfill
25 \pspicture(7,6)\psgrid[subgriddiv=0]
26 \SpecialCoor
27 \pnode(5,2){Node}
28 \PstRandomLine[arrowscale=3,showpoints=true,dotscale=2]
29 {->}(2,4)(3,3)(Node)(6,5)
30 \PstRandomLine[linestyle=dashed,RandomCoefficient=2]
31 (2,4)(3,3)(Node)(6,5)
32 \PstRandomLine[linestyle=dotted,RandomCoefficient=0.5]
33 (2,4)(3,3)(Node)(6,5)
34 \PstRandomLine[xunit=0.5,linecolor=red,showpoints=true,
35 dotstyle=square,dotscale=2,
36 RandomCoefficient=3](2,4)(3,3)(Node)(6,5)
37 \endpspicture

```



Closed objects will be illustrated by a variation of the \pspolygon nmacro.

After dealing with the optional parameters, the \PstRandomPolygon@i macro initiates the closed object by calling the \begin@ClosedObj macro, then reading and accumulating the coordinates by the \pst@getcoors macro as in the preceding randomized line example. Then the \PstRandomPolygon@ii macro calls the Polygon PostScript operator (using the \tx@Polygon T_{EX} macro), which is defined in the PostScript header file pstricks.pro. It uses a Lineto macro, which will take to be the same as in the previous example, using the macro \PstRandomLine@iii. To finish, the type of the line is set by the **linetype** parameter as a “closed curve with no particular symmetry”, and finally the \end@ClosedObj macro is called.

```

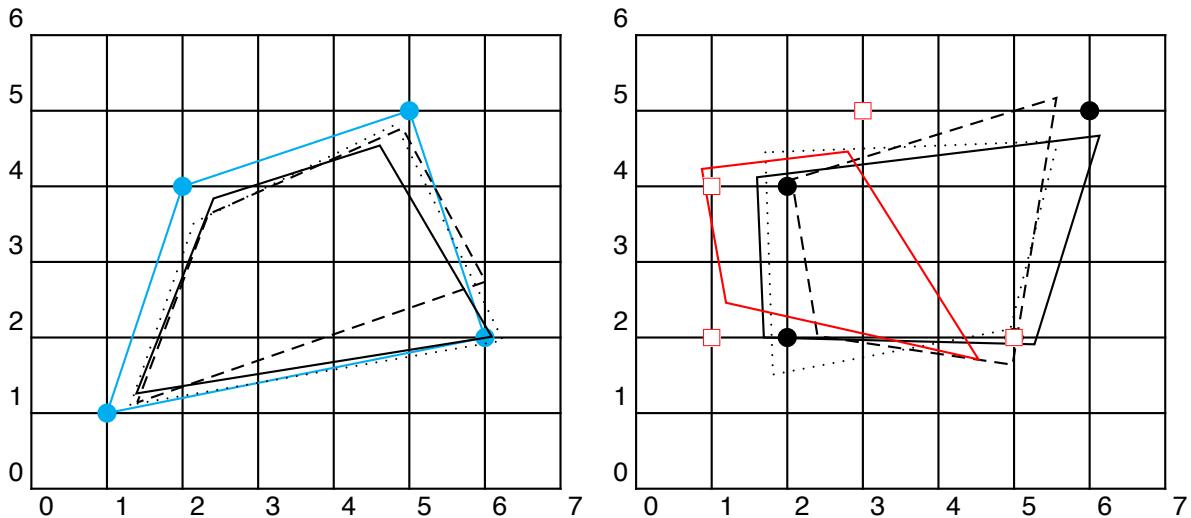
1 \def\PstRandomPolygon{\pst@object{PstRandomPolygon}}
2
3 \def\PstRandomPolygon@i{%
4 \begin@ClosedObj

```

```

5  \def\pst@cp{}%
6  \pst@getcoors{\PstRandomPolygon@ii}%
7
8  \def\PstRandomPolygon@ii{%
9    \addto@pscode{\PstRandomLine@iii \tx@Polygon}%
10   \def\pst@linetype{1}%
11   \end@ClosedObj}%
12
13 \pspicture(7,6)\psgrid[subgriddiv=0]
14   \pspolygon[linecolor=cyan,showpoints=true,dotscale=2]
15     (1,1)(2,4)(5,5)(6,2)
16   \PstRandomPolygon(1,1)(2,4)(5,5)(6,2)
17   \PstRandomPolygon[linestyle=dashed](1,1)(2,4)(5,5)(6,2)
18   \PstRandomPolygon[linestyle=dotted](1,1)(2,4)(5,5)(6,2)
19 \endpspicture
20 \hfill
21 \pspicture(7,6)\psgrid[subgriddiv=0]
22   \SpecialCoor
23   \pnode(5,2){Node}
24   \PstRandomPolygon[showpoints=true,dotscale=2](2,4)(2,2)(Node)
25     (6,5)
26   \PstRandomPolygon[linestyle=dashed,RandomCoefficient=2]
27     (2,4)(2,2)(Node)(6,5)
28   \PstRandomPolygon[linestyle=dotted,RandomCoefficient=0.5]
29     (2,4)(2,2)(Node)(6,5)
30   \PstRandomPolygon[xunit=0.5,linecolor=red,showpoints=true,
31     dotstyle=square,dotscale=2,
32     RandomCoefficient=3](2,4)(2,2)(Node)(6,5)
33 \endpspicture

```



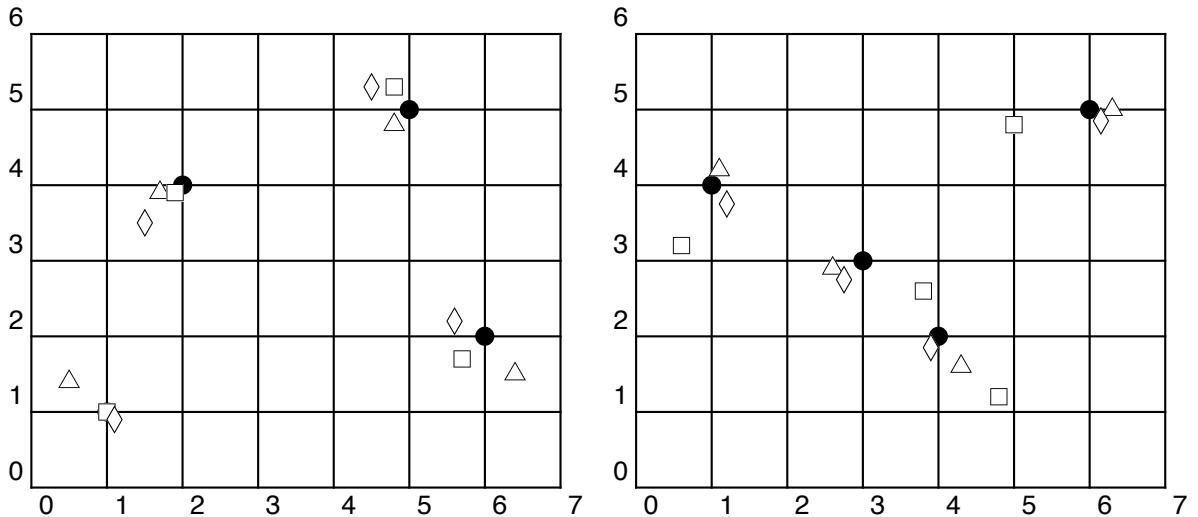
Special objects will be illustrated by a variation of the `\psdots` macro. After dealing with the optional parameters, the `\PstRandomDots@i` macro initiates the special object by calling the `\begin@SpecialObj` macro, then reading and accumulating the coordinates by the `\pst@getcoors` macro, as in the randomized line example above. Then the `\PstRandomDots@ii` macro calls the `NArray` PostScript operator (using the `\tx@NArray` `TEX` macro),

which is defined in the PostScript header file `pstricks.pro`. It handles the *array* of coordinates by applying the `\PstRandomDots@iii` macro, which specifies the style and size to use for the dots and calls the `Dot` PostScript operator, also defined in the header file `pstricks.pro`. As before, we just add a random noise to the coordinates before calling the macro which typesets the dot. Finally, the `\end@SpecialObj` macro is called.

```

1 \def\PstRandomDots{\pst@object{PstRandomDots}}
2
3 \def\PstRandomDots@#1{%
4 \begin@SpecialObj
5 \pst@getcoors{\PstRandomDots@#1}
6
7 \def\PstRandomDots@#1{%
8 \addto@pscode{false \tx@NArray \PstRandomDots@#1}%
9 \end@SpecialObj}
10
11 \def\PstRandomDots@#1{%
12 \psk@dotsize
13 \@nameuse{psds@\psk@dotstyle}
14 newpath
15 /RandomValue {rand 10 mod 10 div 0.5 sub
16           \pst@number{\psunit} mul
17           \PstRandomLine@RandomCoefficient\space mul} def
18 n { transform floor 0.5 add exch floor 0.5 add exch itransform
19       exch RandomValue add exch RandomValue add Dot } repeat}
20
21 \pspicture(7,6)\psgrid[subgriddiv=0]
22 \psset{dotscale=2}%
23 \psdots(1,1)(2,4)(5,5)(6,2)
24 \PstRandomDots[dotstyle=triangle](1,1)(2,4)(5,5)(6,2)
25 \PstRandomDots[dotstyle=square](1,1)(2,4)(5,5)(6,2)
26 \PstRandomDots[dotstyle=diamond](1,1)(2,4)(5,5)(6,2)
27 \endpspicture
28 \hfill
29 \pspicture(7,6)\psgrid[subgriddiv=0]
30 \SpecialCoor
31 \psset{dotscale=2}%
32 \pnode(4,2){Node}
33 \psdots(1,4)(3,3)(Node)(6,5)
34 \PstRandomDots[dotstyle=triangle](1,4)(3,3)(Node)(6,5)
35 \PstRandomDots[dotstyle=square,RandomCoefficient=2]
36   (1,4)(3,3)(Node)(6,5)
37 \PstRandomDots[dotstyle=diamond,RandomCoefficient=0.5]
38   (1,4)(3,3)(Node)(6,5)
39 \endpspicture

```



High-level objects

New *high-level* objects, which can also use coordinates, are much more frequently needed than new *low-level* objects in new developments.

The first illustration is a high-level object used to draw a sequence of dots, each having a different scale. Of course, we can use a list of `\psdot[scale=?](x0,y0)` macros, but if we want to have a macro to handle such a case in one call, we must do something like the `\PstDotsScaled` macro. After dealing with the optional parameters, the `\PstDotsScaled@i` macro creates a group, to ensure that the change of parameters will be local to the macro, activates the modified parameters, then calls the `\PstDotsScaled@GetCoordinate` macro without argument. This one will read the two next arguments, one as a coordinate inside parentheses and one as a scale value inside brackets. Here we have to avoid reading two values for each coordinate, as this would prohibit the usage of *special* ones and prevent managing the scaling factor as the third value of the same set. So, it is clear that a syntax like:

```
1 \PstDotsScaled(x0,y0,1.5)(x1,y1,2)(x2,y2,5)(x3,y3,2.8)
```

which could seem easier to read, must be avoided.

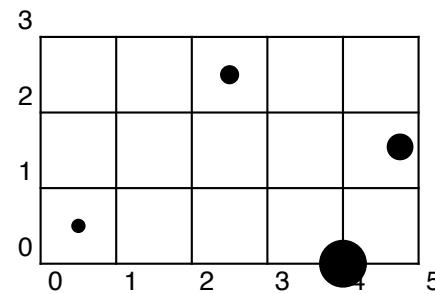
The `\@ifnextchar` macro allows determining if there is another coordinate to manage by testing if the next character to read is an open parenthesis. If it is, the `\PstDotsScaled@GetCoordinate` macro is called again, otherwise the opened group is closed. This is a simple and powerful mechanism to deal with an unknown number of arguments.

```
1 \def\PstDotsScaled{\def\pst@par{}\pst@object{PstDotsScaled}}
2
3 \def\PstDotsScaled@i{%
4 \begingroup
5 \use@par% Assignment of local parameters
6 \PstDotsScaled@GetCoordinate}
```

```

7  \def\PstDotsScaled@GetCoordinate(#1)#2{%
8  \psdot[dotscale=#2](#1)
9  \ifnextchar({\PstDotsScaled@GetCoordinate}{\endgroup}}
10
11 \pspicture(5,3)\psgrid[subgriddiv=0]
12   \SpecialCoor
13   \pnode(2.5,2.5){Node}
14   % "rand 5 mod" return an integer between 0 and 5 and
15   % "rand 3 mod" return an integer between 0 and 3
16   \PstDotsScaled(0.5,0.5){1.5}(Node){2}
17   (! rand 5 mod rand 3 mod){5}(5;18){2.8}
18
19 \endpspicture

```



In the same way, we will now define a macro to draw a series of dots and an error bar for each of them, using two additional parameters.

```

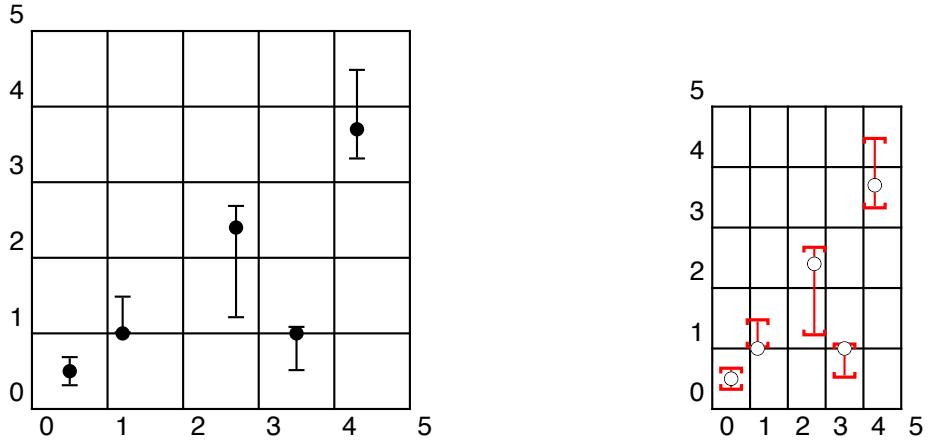
1  % Style of error bars parameter
2  \define@key{psset}{StyleBars}{%
3  \edef\PstDataWithErrorBars@StyleBars#1}%
4
5  \newpsstyle[StyleBars@Default]{arrows=l-l}
6  \setkeys{psset}{StyleBars=StyleBars@Default}% Default value
7
8  \def\PstDataWithErrorBars{%
9  \def\pst@par{} \pst@object{\PstDataWithErrorBars}}%
10
11 \def\PstDataWithErrorBars@#1{%
12 \begingroup
13 \use@par% Assignment of local parameters
14 \PstDataWithErrorBars@GetCoordinate}%
15
16 \def\PstDataWithErrorBars@GetCoordinate(#1,#2){%
17 \PstDataWithErrorBars@DoCoordinate(#1,#2){#3}{#4}}%
18 \ifnextchar({\PstDataWithErrorBars@GetCoordinate}{\endgroup}}%
19
20 \def\PstDataWithErrorBars@DoCoordinate(#1,#2){%
21 \psline[style=\PstDataWithErrorBars@StyleBars]{%
22 ! #1 #2 #3 add}(! #1 #2 #4 add)}%
23 \psdot(#1,#2)}%
24
25 \SpecialCoor
26

```

```

27 \psset{dotscale=1.5}%
28
29 \pspicture(5,5)\psgrid[subgriddiv=0]
30 \PstDataWithErrorBars(0.5,0.5){-0.2}{0.2}(1.2,1){0}{0.5}
31 (2.7,2.4){-1.2}{0.3}(3.5,1){-0.5}{0.1}(4.3,3.7){-0.4}{0.8}
32 \endpspicture
33 \hfill
34 %
35 \newpsstyle{MyStyleBars}{linecolor=red,arrows=[-],arrowscale=1.5}
36 \psset{xunit=0.5,yunit=0.8}%
37 \pspicture(5,5)\psgrid[subgriddiv=0]
38 \PstDataWithErrorBars[dotstyle=o,StyleBars=MyStyleBars]
39 (0.5,0.5){-0.2}{0.2}(1.2,1){0}{0.5}
40 (2.7,2.4){-1.2}{0.3}(3.5,1){-0.5}{0.1}(4.3,3.7){-0.4}{0.8}
41 \endpspicture

```



Here, we deal with each coordinate as two parameters separated by a comma, so this will work only for cartesian coordinates. To manage *special* ones, we must of course use only one argument, then call the `\pst@getcoor` macro which will return the two coordinates of the point, even if it is a node or the result of a PostScript computation. But we must rescale these values at the `\TeX` level, as the error values are defined in the `\TeX` world unit system and the result of `\pst@getcoor` at the PostScript one. This is why we divide the horizontal coordinate by `\psxunit` and the vertical one by `\psyunit`. And to avoid computing them twice, we store these values in PostScript variables, using the `\pst@Verb` macro.

```

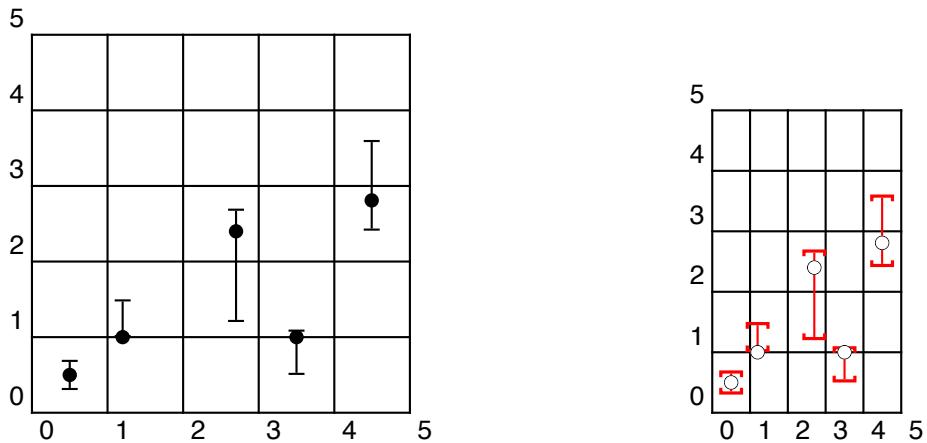
1 \def\PstDataWithErrorBars@GetCoordinate(#1)#2#3{%
2 \PstDataWithErrorBars@DoCoordinate(#1){#2}{#3}%
3 \@ifnextchar({\PstDataWithErrorBars@GetCoordinate}{\endgroup}%
4
5 \def\PstDataWithErrorBars@DoCoordinate(#1){#2}{#3}{%
6 \pst@getcoor{#1}{\pst@tempb}%
7 \pst@Verb{%
8 /XCoor {\pst@tempb pop \pst@number{\psxunit} div} def
9 /YCoor {\pst@tempb exch pop \pst@number{\psyunit} div} def}%
10 \psline[style=\PstDataWithErrorBars@StyleBars]

```

```

11      (! XCoor YCoor #2 add)(! XCoor YCoor #3 add)
12 \psdot(#1}
13
14 \SpecialCoor
15
16 \psset{dotscale=1.5}%
17
18 \pspicture(5,5)\psgrid[subgriddiv=0]
19   \pnode(2.7,2.4){Node}
20   \PstDataWithErrorBars(0.5,0.5){-0.2}{0.2}{(1.2,1){0}{0.5}}
21     (Node){-1.2}{0.3}{! 2 1.5 add 1}{-0.5}{0.1}
22     (5.3,32){-0.4}{0.8}
23 \endpspicture
24 \hfill
25 %
26 \newpsstyle{MyStyleBars}{linecolor=red,arrows=[-],arrowscale=1.5}
27 \psset{xunit=0.5,yunit=0.8}%
28 \pspicture(5,5)\psgrid[subgriddiv=0]
29   \pnode(2.7,2.4){Node}
30   \PstDataWithErrorBars[dotstyle=o,StyleBars=MyStyleBars]
31     (0.5,0.5){-0.2}{0.2}{(1.2,1){0}{0.5}}(Node){-1.2}{0.3}
32     (! 2 1.5 add 1){-0.5}{0.1}
33     (! 5.3 32 cos mul 5.3 32 sin mul){-0.4}{0.8}
34 \endpspicture

```



Sometimes, we need to plot data produced by external programs. There are several ways to handle this. One way is to require that these files be written with some specific \TeX instructions, then use the `\input` macro or directly code the input/output operations using the appropriate \TeX macros. An easier way uses raw data files (the `err-bars.dat` file that we will use in the next example):

```

1 0.7 4.1 -0.7 0.6
2 2.6 3.4 -2.1 0.2
3 3.2 2 -1.2 0
4 4.1 4.6 -0.3 0.2
5 4.6 1.4 -0.5 1.8

```

or

```
1 (0.7, 4.1, -0.7, 0.6)
2 (2.6, 3.4, -2.1, 0.2)
3 (3.2, 2, -1.2, 0)
4 (4.1, 4.6, -0.3, 0.2)
5 (4.6, 1.4, -0.5, 1.8)
```

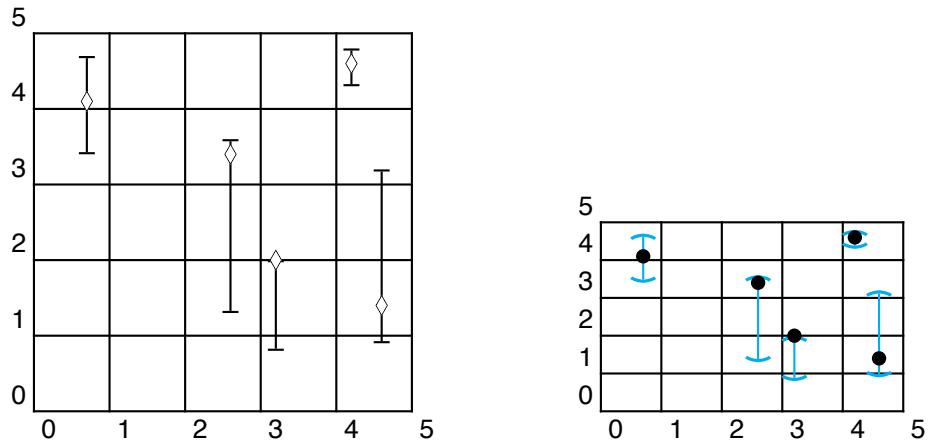
and use the `\readdata` macro (see the Section 11) to store them in a personal macro (note that only cartesian coordinates can be used in this context.) This will produce a macro with the values separated by a D symbol. We can either use this macro like it as a PostScript argument, defining the D operator to do what we want with the coordinates (this is how the plotting commands like `\psplot` work), or parse this macro to extract the coordinates. This is what we will do in the next example.

```
1 \def\PstDataFileWithErrorBars{%
2 \def\pst@par{\pst@object{PstDataFileWithErrorBars}}
3 %
4 \def\PstDataFileWithErrorBars@#1{%
5 \begingroup
6 \use@par% Assignment of local parameters
7 % Read the data from the file and put them in a macro
8 \readdata{\PstDataFileWithErrorBars@Data}{#1}%
9 \PstDataFileWithErrorBars@GetCoordinate{%
10   \PstDataFileWithErrorBars@Data}}
11 %
12 \def\PstDataFileWithErrorBars@GetCoordinate#1{%
13 \expandafter\PstDataFileWithErrorBars@GetCoordinate@#1 }
14 %
15 \def\PstDataFileWithErrorBars@GetCoordinate@#1{%
16 \PstDataFileWithErrorBars@GetCoordinate@ii#1}
17 %
18 \def\PstDataFileWithErrorBars@GetCoordinate@ii{%
19   #1 #2 #3 #4 #5 #6 #7 #8 {%
20 \PstDataFileWithErrorBars@DoCoordinate(#2,#4){#6}{#8}%
21 \@ifnextchar D{\PstDataFileWithErrorBars@GetCoordinate@ii}{%
22   \endgroup}}
23 %
24 \let\PstDataFileWithErrorBars@DoCoordinate
25   \PstDataFileWithErrorBars@DoCoordinate
26 %
27 \psset{dotscale=1.5}%
28 %
29 \pspicture(5,5)\psgrid[subgriddiv=0]
30   \PstDataFileWithErrorBars[dotstyle=diamond]{err-bars.dat}
31 \endpspicture
32 \hfill
33 %
34 \newpsstyle{MyStyleBars}{linecolor=cyan,arrows=(-),arrowscale=1.5}
```

```

35 \psset{xunit=0.8,yunit=0.5}
36 \pspicture(5,5)\psgrid[subgriddiv=0]
37 \PstDataFileWithErrorBars[StyleBars=MyStyleBars]{err-bars.dat}
38 \endpspicture

```



We finish this section with a more sophisticated example, which draws a general mesh based on triangles, as used in finite elements numerical methods, with an automatic computation of the barycenters of the triangles to position their numbers. We want of course to be as generic as possible, just defining here the points of the mesh (with any number of them, and possibly defined as *special* coordinates), then the list of the triplets of these points which form the triangles.

Note that we store the definition of the points as macros, whose names are suffixed by the rank of the point in the list of coordinates. We do not need here to store them as PSTricks nodes, which would make the resulting PostScript code larger and slower without any benefit.

```

1 \def\PstMeshNodes{%
2 \pst@cnth=\z@
3 \PstMeshNodes@GetCoordinate}
4
5 \def\PstMeshNodes@GetCoordinate(#1){%
6 \advance\pst@cnth\@ne
7 \namedef{MeshNode}{\the\pst@cnth}{#1}%
8 \@ifnextchar(\PstMeshNodes@GetCoordinate){}}
9
10 \def\PstMeshTriangles{\def\pst@par{}\pst@object{PstMeshTriangles}}
11
12 \def\PstMeshTriangles@#1{%
13 \begingroup
14 \use@par% Assignment of local parameters
15 % PostScript macro to help to compute the barycenter
16 % of the three points
17 \pst@Verb{/ForBarycenter {add 2 div exch dup /X3 ED sub
18 4 1 roll
19 add 2 div neg exch dup /X6 ED

```

```

20   add div dup X6 mul X3 add} def}%
21 \PstMeshTriangles@GetTriangle}

22
23 \def\PstMeshTriangles@GetTriangle#1#2#3#4{%
24 % #1, #2, #3 : first, second and third point numbers
25 % #4 : label to print on the barycenter of this triangle
26 \pspolygon{@nameuse{MeshNode#1}}{@nameuse{MeshNode#2}}
27   (@nameuse{MeshNode#3})
28 \PstMeshPrintNode{@nameuse{MeshNode#1}}#1%
29 \PstMeshPrintNode{@nameuse{MeshNode#2}}#2%
30 \PstMeshPrintNode{@nameuse{MeshNode#3}}#3%
31 \PstMeshTrianglesLabelOnBarycenter{@nameuse{MeshNode#1}}
32   {@nameuse{MeshNode#2}}{@nameuse{MeshNode#3}}#4}%
33 \@ifnextchar(\PstMeshTriangles@GetTriangle\endgroup}

34
35 \def\PstMeshTrianglesLabelOnBarycenter#1#2#3#4{%
36 % #1 : coordinates of the first point (A) of the triangle
37 % #2 : coordinates of the second point (B) of the triangle
38 % #3 : coordinates of the third point (C) of the triangle
39 % #4 : label to print on the barycenter
40 \pst@getcoor{#1}\pst@tempa}%
41 \pst@getcoor{#2}\pst@tempb}%
42 \pst@getcoor{#3}\pst@tempc}%
43 \rput(! % To retrieve the X and Y coordinates of the three points
44   \pst@tempa \pst@number{\psyunit} div /YA ED
45   \pst@number{\psxunit} div /XA ED
46   \pst@tempb \pst@number{\psyunit} div /YB ED
47   \pst@number{\psxunit} div /XB ED
48   \pst@tempc \pst@number{\psyunit} div /YC ED
49   \pst@number{\psxunit} div /XC ED
50   XA XB add 2 div XC eq
51     {XA XC XB YA YC YB ForBarycenter}
52     {XC XB XA YC YB YA ForBarycenter} ifelse
53   /ZB ED /ZA ED
54   XA XC add 2 div XB eq
55     {XA XC XB YA YC YB ForBarycenter}
56     {XB XC XA YB YC YA ForBarycenter} ifelse
57   /ZD ED /ZC ED
58   %  $X = (ZB - ZD) / (ZA - ZC)$  and  $Y = ZD - ZC * X$ 
59   ZB ZD sub ZA ZC sub div dup ZC mul neg ZD add)
60   {\PstMeshPrintLabel{#4}}}

61
62 % Default format to print the mesh nodes
63 \def\PstMeshPrintNode#1#2{\rput*{#1}{\bf\scalebox{2}{#2}}}

64
65 % Default format to print the mesh labels on the barycenters
66 \def\PstMeshPrintLabel#1{\it\scalebox{2}{#1}}

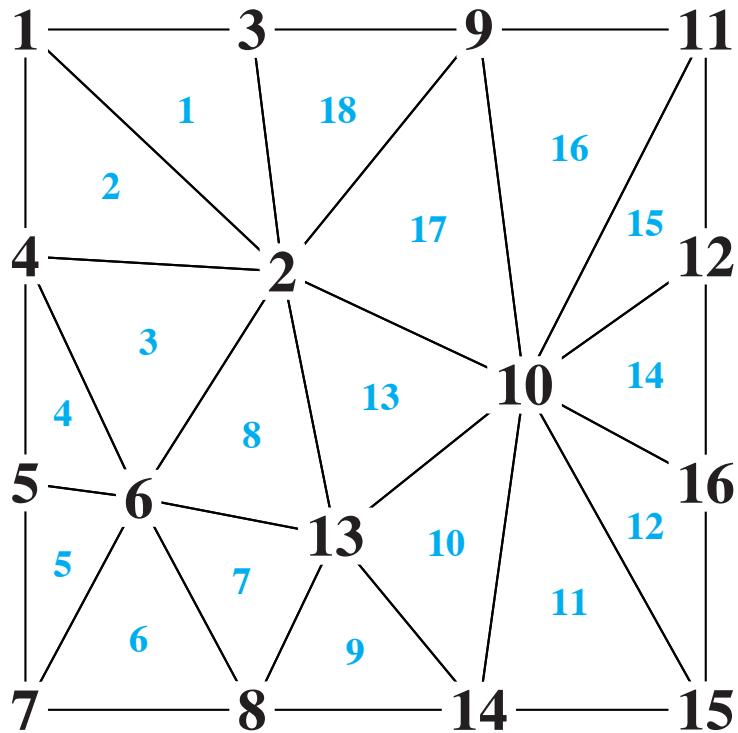

67
68 % Usage examples
69 % =====
70
71 % Definition of the nodes of the mesh
72 \PstMeshNodes(0,9)(3.4,5.8)(3,9)(0,6)(0,3)(1.5,2.8)(0,0)(3,0)

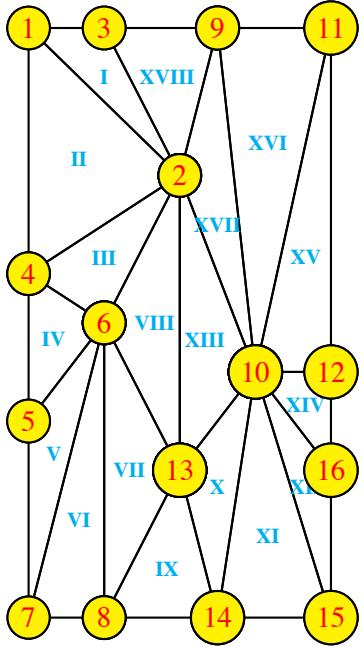
```

```

73      (6,9)(6.6,4.3)(9,9)(9,6)(4.1,2.3)(6,0)(9,0)(9,3)
74
75 % To draw the mesh
76 \newcommand{\PstMeshTrianglesDraw}[1][]{%
77 \PstMeshTriangles[#1](1,2,3){1}(1,2,4){2}(2,4,6){3}(4,5,6){4}(5,6,7){5}
78   (6,7,8){6}(6,8,13){7}(2,6,13){8}(8,13,14){9}(10,13,14){10}
79   (10,14,15){11}(10,15,16){12}(2,10,13){13}(10,12,16){14}
80   (10,11,12){15}(9,10,11){16}(2,9,10){17}(2,3,9){18}}
81
82 % New format for the label
83 \renewcommand{\PstMeshPrintLabel}[1]{%
84   \Large\textbf{\textcolor{cyan}{#1}}}
85
86 \PstMeshTrianglesDraw

```





```

1 % New definition of the nodes of the mesh
2 \PstMeshNodes(0,12)(4,9)(2,12)(0,7)(0,4)(2,6)(0,0)(2,0)
3 (5,12)(6,5)(8,12)(8,5)(4,3)(5,0)(8,0)(8,3)
4
5 \renewcommand{\PstMeshPrintNode}[2]{%
6 \rput(#1){%
7 \pscirclebox[fillstyle=solid,fillcolor=yellow]{\textcolor{red}{#2}}}}
8
9 \makeatletter
10 \renewcommand{\PstMeshPrintLabel}[1]{%
11 \scriptsize\textbf{\textcolor{cyan}{\@Roman{#1}}}}
12 \makeatother
13
14 % We use the mesh previously defined
15 \PstMeshTrianglesDraw[xunit=0.5,yunit=0.65]

```

63.6 Generic template for a package

A generic template for a new specialized package should have the following form:⁵¹

```

1 \def\FileVersion{1.2}
2 \def\FileDate{2003/06/22}
3
4 % To identify the package when it will be loaded
5 \message{'pst-xxxx' \FileVersion, \FileDate\space
6 (FirstName LastName)}
7
8 % To allow other packages to test if this one
9 % was already loaded
10 \csname PstXxxxLoaded\endcsname
11 \let\PstXxxxLoaded\endinput
12
13 % Require of course the PSTricks package,
14 % and perhaps other ones
15 \ifx\PSTricksLoaded\endinput\else\input{pstricks}\fi
16 %\ifx\PSTnodesLoaded\endinput\else\input{pst-node}\fi
17 %\ifx\MultidoLoaded\endinput\else\input{multido}\fi
18
19 % David Carlisle interface to the 'keyval' package

```

⁵¹For a short, but complete and pedagogically useful example, you can look at the commented source code of the ‘pst-li3d’ package [62], also described in Section 64.5, which allow to add a lighten effect to text and curves, or to the ‘pst-poly’ package [63], which is used to draw polygons.

```

20 \input{pst-key}
21
22 % Allow to use the @ character for all internal macros
23 \edef\PstAtCode{\the\catcode`\@}
24 \catcode`\@=11\relax
25
26 % Definition of the specific parameters
27 % =====
28
29 % A parameter containing an integer value
30 \define@key{psset}{Integer}{\pst@getint{\#1}\PstXxxx@Integer}}
31
32 % A parameter containing a real value
33 \define@key{psset}{Real}{\pst@checknum{\#1}\PstXxxx@Real}}
34
35 % A parameter containing a string value
36 \define@key{psset}{String}{\edef\PstXxxx@String{\#1}}
37
38 % A parameter containing a boolean value
39 \newif\ifPstXxxx@Boolean
40 \define@key{psset}{Boolean}[true]{%
41 \@nameuse{PstXxxx@Boolean#1}}
42
43 % Other parameters
44 % ...
45
46 % Defaults values for these parameters
47 % =====
48 \setkeys{psset}{Integer=10,Real=123.456,String=,Boolean=false}
49
50 % The macro \PstXxxx, with or without optional parameters
51 % =====
52 \def\PstXxxx{\def\pst@par{}\pst@object{PstXxxx}}
53
54 \def\PstXxxx{%
55   %% Two braces, because parameter
56   %% assignments must be local
57   \use@par% Assignment of local parameters
58   % The code itself, which can of course include calls
59   % to auxiliary macros
60   %% ...
61 }
62
63 % Restore the definition of the @ character
64 \catcode`\@=\PstAtCode\relax
65
\endinput

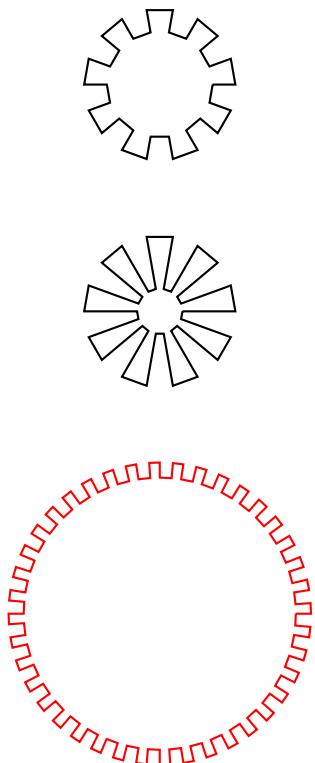
```

64 Examples

This section gives five examples which illustrate the major features of the ‘pst-key’ package and show how to use it to write high-level macros and packages. These examples are explained in step by step detail, trying to illustrate many different techniques which can be put to use by developers.

64.1 Basic example : cogged wheels

First, a simple example to define “cogged wheels”. The crucial macro is first developed, than converted afterwards into a PSTricks object.



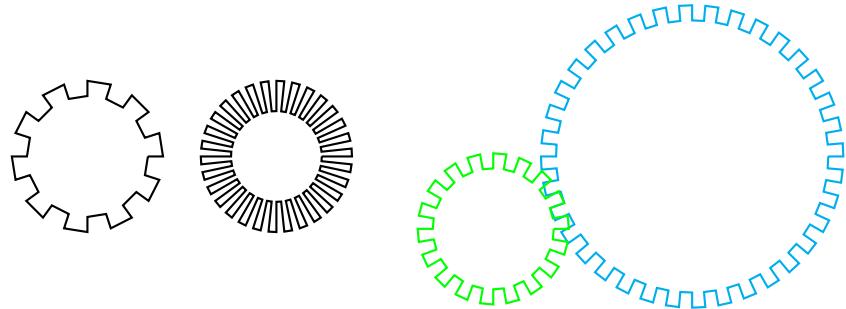
```
1 \SpecialCoor
2
3 \def\PstCoggedWheel#1#2{%
4 % #1 = number of teeth (it must be a divisor of 360!)
5 % #2 = height of the teeth, between 0 and 1 unit
6 \pst@cntc=360
7 \divide\pst@cntc by #1
8 \pst@cntd=\pst@cntc
9 \divide\pst@cntd\tw@
10 \pst@dimd=\psunit
11 \pssetlength{\pst@dimc}{#2}%
12 \advance\pst@dimd-\pst@dimc
13 \pscustom{%
14   \moveto(\pst@dimd;0)
15   \multido{\iAngleA=1+1,\pst@cntc,
16           \iAngleB=\pst@cntd+\pst@cntc,
17           \iAngleC=\pst@cntc+\pst@cntc}{#1}{%
18     \lineto(1;\iAngleA)
19     \lineto(1;\iAngleB)
20     \lineto(\pst@dimd;\iAngleB)
21     \lineto(\pst@dimd;\iAngleC)}}
22
23 \rput(-2,4){\PstCoggedWheel{9}{0.3}}
24 \rput(-2,1){\PstCoggedWheel{9}{7mm}}
25 \rput(-2,-3){\psset{unit=2,linecolor=red}\PstCoggedWheel{40}{0.1}}
```

```
1 % "TeethNumber": number of teeth (it must be a divisor of 360!)
2 \define@key{psset}{TeethNumber}{%
3   \pst@getint{#1}{\PstCoggedWheel@TeethNumber}}
4
5 % "TeethHeight": height of the teeth, in current unit
6 \define@key{psset}{TeethHeight}{%
7   \edef\PstCoggedWheel@TeethHeight{#1}}
8
9 % Default parameters values
10 \setkeys{psset}{TeethNumber=10,TeethHeight=0.2}
11
12 % Main macro for "cogged wheel" object
```

```

13 \def\PstCoggedWheel{\def\pst@par{}\pst@object{PstCoggedWheel}}
14
15 \def\PstCoggedWheel@#1{%
16   \use@par% Assignment of local parameters
17   \pst@cntc=360
18   \divide\pst@cntc\PstCoggedWheel@TeethNumber
19   % Test to verify if the number of teeth is valid
20   \pst@cnth=\pst@cntc
21   \multiply\pst@cnth\PstCoggedWheel@TeethNumber
22   \ifnum\pst@cnth=360
23   \else
24     \@pstrickserr{TeethNumber }{\PstCoggedWheel@TeethNumber}
25     must be a divisor of 360! Results will be inaccurate.}{\@ehpb}%
26 \fi
27 \pst@cntd=\pst@cntc
28 \divide\pst@cntd\tw@
29 \pst@dimd=\psunit
30 \pssetlength{\pst@dimc}{\PstCoggedWheel@TeethHeight}%
31 \advance\pst@dimd-\pst@dimc
32 \pscustom{%
33   \moveto(\pst@dimd;0)
34   \multido{\iAngleA=z@+\pst@cntc,\iAngleB=\pst@cntd+\pst@cntc,
35         \iAngleC=\pst@cntc+\pst@cntc}{%
36     \PstCoggedWheel@TeethNumber}%
37   \lineto(1;\iAngleA)
38   \lineto(1;\iAngleB)
39   \lineto(\pst@dimd;\iAngleB)
40   \lineto(\pst@dimd;\iAngleC)}}
41
42 \rput(-4,0){\PstCoggedWheel}
43 \rput(-1.5,0){\PstCoggedWheel[TeethNumber=30,TeethHeight=4mm]}
44 \rput(4,0){%
45   \PstCoggedWheel[unit=2,linecolor=cyan,
46                   TeethNumber=36,TeethHeight=0.1]%
47   \rput(2.8;200){%
48     \PstCoggedWheel[linecolor=green,TeethNumber=18]}}

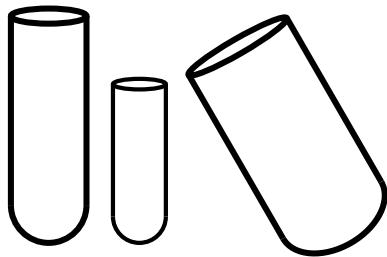
```



64.2 Advanced example: test tube

This is a complex example which defines a high level macro to draw a test tube, as used in chemistry.⁵²

We first define a simple tube tube. Note that we use the macro `\psellipticarc` rather than `\psarc(0,0.5){0.5}{180}{0}` because it scales correctly if we change the **xunit** parameter. The `\psarc` macro scales only with the **runit** parameter, which is not a logical parameter to use here.

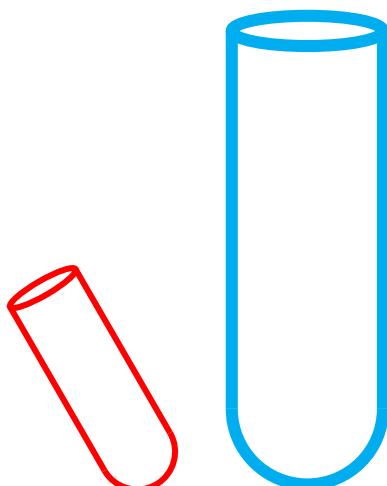


```

1 \def\PstTestTube{%
2   % Two braces to limit the scope of the
3   % next settings inside the macro only
4   \psset{dimen=middle,linewidth=0.08}%
5   \psline(-0.5,3)(-0.5,0.5)
6   \psellipticarc(0,0.5)(0.5,0.5){180}{0}
7   \psline(0.5,0.5)(0.5,3)
8   \psellipse(0,3)(0.5,0.1)}%
9
10 \rput(-4,0){\PstTestTube}
11 \rput(-2.8,0){\psset{unit=0.7}\PstTestTube}
12 \rput(30){\psset{xunit=1.5}\PstTestTube}

```

To be able to accept later various options, we will transform this macro to allow it to accept optional parameters, as already demonstrated.



```

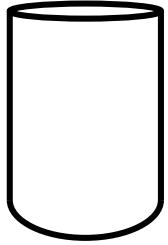
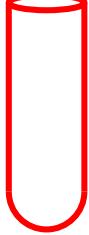
% Macro for tube (\PstTestTube last macro)
1 \def\PstLaboTestTube@Tube{%
2   \psset{linewidth=0.08}%
3   \psline(-0.5,3)(-0.5,0.5)
4   \psellipticarc(0,0.5)(0.5,0.5){180}{0}
5   \psline(0.5,0.5)(0.5,3)
6   \psellipse(0,3)(0.5,0.1)}%
7
8 % Main macro for "test tube" object
9 \def\PstLaboTestTube{%
10   \def\pst@par{} \pst@object{\PstLaboTestTube}}%
11
12 \def\PstLaboTestTube@#1{%
13   \setkeys{psset}{dimen=middle}%
14   \use@par% Assignment of local parameters
15   \PstLaboTestTube@Tube}%
16
17 \rput{30}{\PstLaboTestTube[linecolor=red]}
18 \rput{-10}{\PstLaboTestTube[unit=2,linecolor=cyan]}
19

```

As we have seen, we can use as usual the `\rput` macro to put this objects at specified locations, with an optional rotation. But if we want to be able to

⁵²This is extracted and adapted from the ‘pst-labo’ package [61] by Manuel Luque and Christophe Jorssen, which define a lot more objects and parameters.

specify a coordinate as an optional parameter of the macro (with $(0,0)$ for default, as usual), this is easy to do, using what was previously explained. Nevertheless, note that between `\PstLaboTestTube[unit=2](x0,y0)` and `\rput(x0,y0){\PstLaboTestTube}` there is a difference, because in the first case the change of unit apply to the coordinates too, and not it the second case. It is not useful to allow specification of an angle here because the result will not be correct if we put a liquid in the tube, as we will do later.



```

1 \def\PstLaboTestTube@i{%
2 % Test for optional coordinate (default = (0,0))
3 \@ifnextchar({\PstLaboTestTube@ii}{%
4 {\PstLaboTestTube@ii(\z@,\z@)})}
5
6 \def\PstLaboTestTube@ii(#1){{%
7 \setkeys{psset}{dimen=middle}%
8 \use@par% Assignment of local parameters
9 \rput(#1){\PstLaboTestTube@Tube}}}
10
11 \PstLaboTestTube[linecolor=red](-3,0)
12 \PstLaboTestTube[xunit=2](0,1)

```

To be able to use several forms for the tubes, we can define a *string* parameter, which can take the different values allowed, here `straight`, `roundbottom`, `erlenmeyer`, `small` (obviously, other ones can be easily added.) The macro `\PstLaboTestTube@Tube` will be defined in an indirect way, as the macro which will define the tube of the chosen form. So, the `\PstLaboTestTube@ii` macro does not have to be modified for the moment. This way of proceeding is very flexible, as it allows adding new forms which are immediately usable without modifying any line of the macros themselves.

We also move the vertical origin of the tube, to be able to keep the same bottom line, with any rotation applied to it (this will require too that the tube will be moved to this vertical position, to ensure that its basis will be at the $(0,0)$ coordinate.)

```

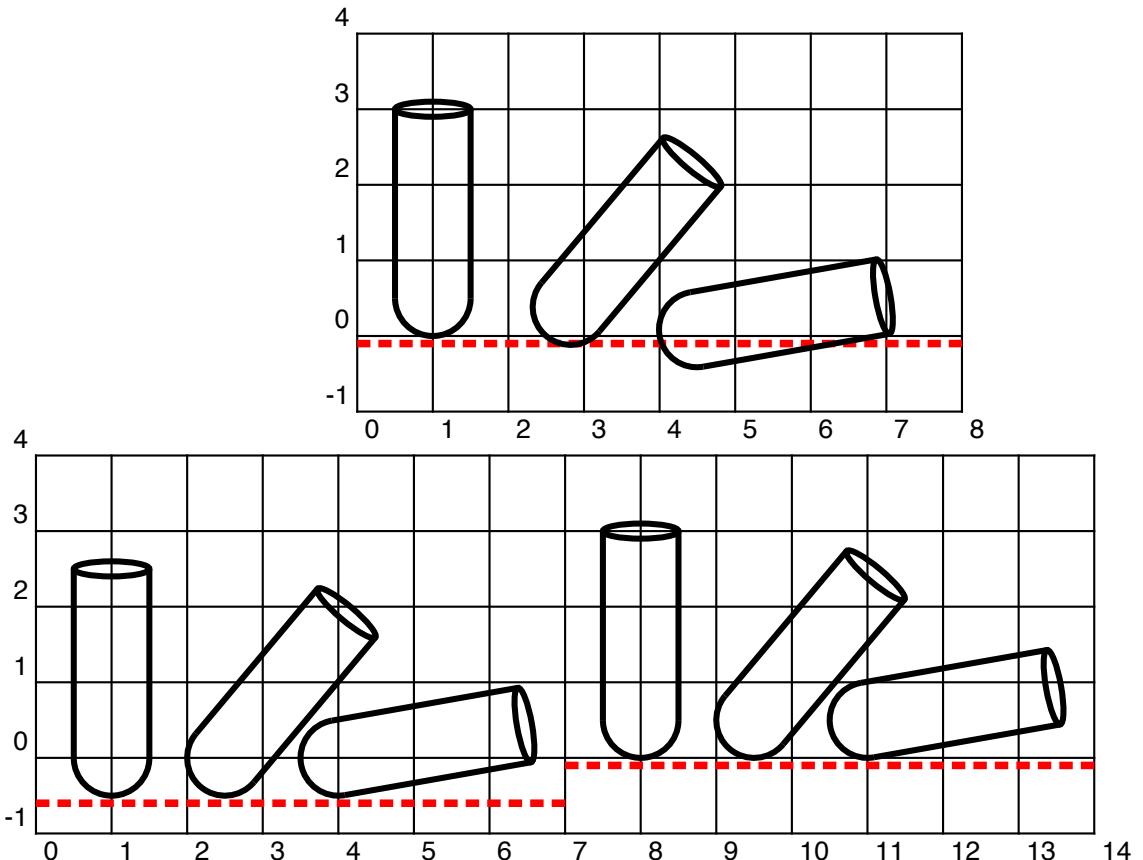
1 \def\PstTestTubeA{%
2 \psset{dimen=middle,linewidth=0.08}%
3 \psline(-0.5,3)(-0.5,0.5)
4 \psellipticarc(0,0.5)(0.5,0.5){180}{0}
5 \psline(0.5,0.5)(0.5,3)
6 \psellipse(0,3)(0.5,0.1)}}
7
8 \def\PstTestTubeB{%
9 \psset{dimen=middle,linewidth=0.08,origin={0,0.5}}%
10 \psline(-0.5,3)(-0.5,0.5)
11 \psellipticarc(0,0.5)(0.5,0.5){180}{0}
12 \psline(0.5,0.5)(0.5,3)
13 \psellipse(0,3)(0.5,0.1)}}
14
15 \rput(1,0){\PstTestTubeA}

```

```

16 \rput{-40}(2.5,0){\PstTestTubeA}
17 \rput{-80}(4,0){\PstTestTubeA}
18
19 \rput(1,0){\PstTestTubeB}
20 \rput{-40}(2.5,0){\PstTestTubeB}
21 \rput{-80}(4,0){\PstTestTubeB}
22 \rput{8,0.5}{\PstTestTubeB}
23 \rput{-40}(9.5,0.5){\PstTestTubeB}
24 \rput{-80}(11,0.5){\PstTestTubeB}

```



The definitions of the six macros `\PstLaboTestTube@Height` (the height of the tube), `\PstLaboTestTube@HeightTop` (the height of the bottom part of the tube above the rotation point), `\PstLaboTestTube@HeightBottom` (the height of the bottom part of the tube below the rotation point), `\PstLaboTestTube@WidthBottomHalf` (half the width of the bottom part), `\PstLaboTestTube@WidthTopHalf` (half the width of the top part) and `\PstLaboTestTube@NeckWidth` (half the width of the neck) will be used later (to know at which height to put a stopper, for instance).

We need also to define the macro `\PstLaboTestTube@HeightBottom` when the `Form` parameter is set and not when the macro `\PstLaboTestTube@Tube` is expanded, because we will need to know its value before invoking the macro defining the kind of tube.

```

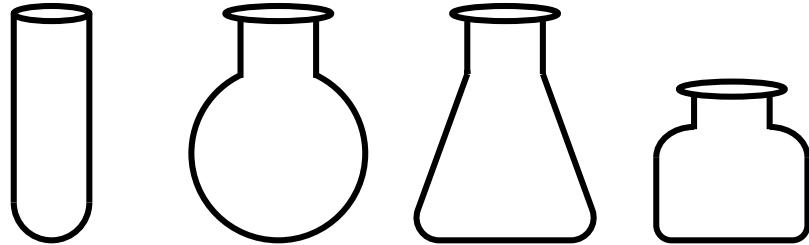
1 % Form parameter
2 \define@key{psset}{Form}{%
3   \def\PstLaboTestTube@Tube{\@nameuse{PstLaboTestTube@#1}}%
4   \def\PstLaboTestTube@HeightBottom{%
5     \@nameuse{PstLaboTestTube@#1@HeightBottom}}}
6
7 \setkeys{psset}{Form=straight}% Default value
8
9 % Macro for straight tube (was \PstLaboTestTube@Tube)
10 \def\PstLaboTestTube@straight{%
11   \gdef\PstLaboTestTube@Height{3}%
12   \gdef\PstLaboTestTube@HeightTop{2.5}%
13   \gdef\PstLaboTestTube@WidthBottomHalf{0.5}%
14   \gdef\PstLaboTestTube@WidthTopHalf{0.5}%
15   \gdef\PstLaboTestTube@WidthNeckHalf{0.5}%
16   \psset{dimen=middle,linewidth=0.08,
17         origin={0,\PstLaboTestTube@straight@HeightBottom}}%
18   \psline(-0.5,3)(-0.5,0.5)
19   \psellipticarc(0,0.5)(0.5,0.5){180}{0}
20   \psline(0.5,0.5)(0.5,3)
21   \psellipse(0,3)(0.5,0.1)}}
22 \def\PstLaboTestTube@straight@HeightBottom{0.5}
23
24 % Macro for round bottom flask
25 \def\PstLaboTestTube@roundbottom{%
26   \gdef\PstLaboTestTube@Height{3}%
27   \gdef\PstLaboTestTube@HeightTop{1.8}%
28   \gdef\PstLaboTestTube@WidthBottomHalf{1.15}%
29   \gdef\PstLaboTestTube@WidthTopHalf{0.5}%
30   \gdef\PstLaboTestTube@WidthNeckHalf{0.7}%
31   \psset{dimen=middle,linewidth=0.08,
32         origin={0,\PstLaboTestTube@roundbottom@HeightBottom}}%
33   \psline(-0.5,2.95)(-0.5,2.15)%
34   \psellipticarc(0,1.15)(1.15,1.15){115}{65}%
35   \psline(0.5,2.15)(0.5,2.95)%
36   \psellipse(0,3)(0.7,0.1)}}
37 \def\PstLaboTestTube@roundbottom@HeightBottom{1.2}
38
39 % Macro for erlenmeyer flask
40 \def\PstLaboTestTube@erlenmeyer{%
41   \gdef\PstLaboTestTube@Height{3}%
42   \gdef\PstLaboTestTube@HeightTop{1.5}%
43   \gdef\PstLaboTestTube@WidthBottomHalf{1.3}%
44   \gdef\PstLaboTestTube@WidthTopHalf{0.5}%
45   \gdef\PstLaboTestTube@WidthNeckHalf{0.7}%
46   \psset{dimen=middle,linewidth=0.08,
47         origin={0,\PstLaboTestTube@erlenmeyer@HeightBottom}}%
48 % Last point required to be able to fill the region correctly
49   \psline(-0.5,2.9)(-0.5,2.2)(-0.5,2.25)
50   \psline[linear=0.3](-0.5,2.2)(-1.3,0)(1.3,0)(0.5,2.2)
51   \psline(0.5,2.2)(0.5,2.9)
52   \psellipse(0,3)(0.7,0.1)}}

```

```

53 \def\PstLaboTestTube@erlenmeyer@HeightBottom{1.5}
54
55 % Macro for small flask
56 \def\PstLaboTestTube@small{\%
57 \gdef\PstLaboTestTube@Height{2}%
58 \gdef\PstLaboTestTube@HeightTop{0.8}%
59 \gdef\PstLaboTestTube@WidthBottomHalf{1}%
60 \gdef\PstLaboTestTube@WidthTopHalf{0.5}%
61 \gdef\PstLaboTestTube@WidthNeckHalf{0.7}%
62 \psset{dimen=middle,linewidth=0.08,%
63 origin={0,\PstLaboTestTube@small@HeightBottom}}%
64 \psline(-0.5,1.9)(-0.5,1.47)
65 \psellipticarc(-0.5,1.1)(0.5,0.4){90}{180}
66 \psline[linearc=0.2]{(-1,1.1)(-1,0)(1,0)(1,1.1)}
67 \psellipticarc(0.5,1.1)(0.5,0.4){0}{90}
68 \psline(0.5,1.47)(0.5,1.9)
69 \psellipse(0,2)(0.7,0.1)}}
70 \def\PstLaboTestTube@small@HeightBottom{1.2}
71
72 \def\PstLaboTestTube@ii(#1){\%
73 \setkeys{psset}{dimen=middle}%
74 \use@par% Assignment of local parameters
75 \rput(#1){%
76 \rput(0,\PstLaboTestTube@HeightBottom){%
77 \PstLaboTestTube@Tube}}}
78
79 \PstLaboTestTube(-4.5,0)
80 \PstLaboTestTube[Form=roundbottom]{(-1.5,0)}
81 \PstLaboTestTube[Form=erlenmeyer]{(1.5,0)}
82 \PstLaboTestTube[Form=small]{(4.5,0)}

```



To be able to add a stopper to the tube, we define a boolean parameter to specify whether a stopper should be drawn or not. We also have to verify (as for all new features) that everything is scaled correctly if the unit is changed, at least for unit, because sometimes changing only xunit or yunit leads to incorrect behavior. In this example, it does work as expected.

```

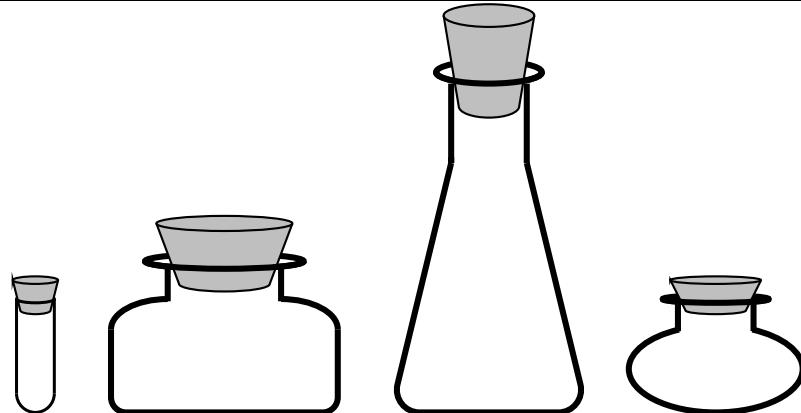
1 % Stopper parameter
2 \newif\ifPstLaboTestTube@Stopper
3 \define@key{psset}{Stopper}[true]{%
4 \nameuse{PstLaboTestTube@Stopper#1}}
5
6 \setkeys{psset}{Stopper=false}% Default value

```

```

7 % Macro for stopper (depends on the height of the tube
8 % and on the width of the neck)
9 \def\PstLaboTestTube@Stopper{%
10 \rput(0,\PstLaboTestTube@HeightTop){%
11   \pscustom[fillstyle=solid,fillcolor=lightgray]{%
12     \psellipticarc(0,0.5)(0.6,0.1){0}{180}
13     \psline(-0.6,0.5)(-0.4,-0.3)
14     \psellipticarc(0,-0.3)(0.4,0.1){180}{0}
15     \psline(0.4,-0.3)(0.6,0.5)}
16     \psellipticarc(0,0.5)(0.6,0.1){180}{0}
17   \psellipticarc*[linewidth=0.08]
18     (\PstLaboTestTube@WidthNeckHalf,0.1){180}{0}}}
19
20
21 \def\PstLaboTestTube@ii(#1){%
22 \setkeys{psset}{dimen=middle}%
23 \use@par% Assignment of local parameters
24 \rput(#1){%
25   \rput(0,\PstLaboTestTube@HeightBottom){%
26     \PstLaboTestTube@Tube
27     \ifPstLaboTestTube@Stopper
28       \PstLaboTestTube@Stopper
29     \fi}}}}
30
31 \psset{Stopper=true}
32 \rput(-4.5,0){\PstLaboTestTube[unit=0.5]}
33 \rput(-2,0){\PstLaboTestTube[xunit=1.5,Form=small]}
34 \rput(1.5,0){\PstLaboTestTube[yunit=1.5,Form=erlenmeyer]}
35 \rput(4.5,0){\PstLaboTestTube[yunit=0.5,Form=roundbottom]}

```



Now, we will add the possibility of having a small escape tube, straight or bent, through the stopper. For this we define a *string* parameter, which can take here the values `straight`, `bent` or `triplebent` (it will be easy to add new ones of any kind), and a *length* parameter (but defined as a macro, as previously explained) which will be the length of the outside part (after the bend for these kinds of escape tubes).

We modify also the `\PstLaboTestTube@Stopper` macro, to allow better visualization of the crossing of the escape tube through the stopper.

```

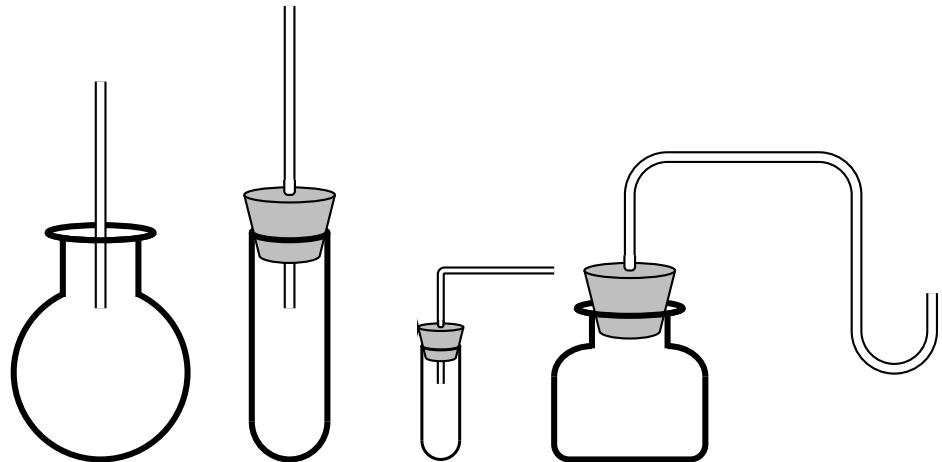
1 % Escape tube type parameter
2 \define@key{psset}{EscapeTube}{%
3 \def\PstLaboTestTube@EscapeTube{%
4 \@nameuse{PstLaboTestTube@EscapeTube@#1}}}
5
6 % Escape tube length parameter
7 \define@key{psset}{EscapeTubeLength}{%
8 \edef\PstLaboTestTube@EscapeTubeLength{\#1}}
9
10 \setkeys{psset}{EscapeTube=,EscapeTubeLength=3}% Defaults
11
12 % Macro for no escape tube
13 \let\PstLaboTestTube@EscapeTube@\relax
14
15 % Macro for straight escape tube
16 \def\PstLaboTestTube@EscapeTube@straight{(%
17 \pssetlength{\pst@dimd}{\PstLaboTestTube@HeightTop}%
18 \psaddtolength{\pst@dimd}{\m@ne}%
19 \rput(0,\pst@dimd){%
20   \psline[doubleline=true,doublesep=0.1]
21     (0,\PstLaboTestTube@EscapeTubeLength)}%
22 \PstLaboTestTube@EscapeTube@AdjustmentsStopper}}
23
24 % Macro for bent escape tube
25 \def\PstLaboTestTube@EscapeTube@bent{(%
26 \pssetlength{\pst@dimd}{\PstLaboTestTube@HeightTop}%
27 \psaddtolength{\pst@dimd}{\m@ne}%
28 \rput(0,\pst@dimd){%
29   \psline[doubleline=true,doublesep=0.1,linearc=0.1]
30     (0,0)(0,3)(\PstLaboTestTube@EscapeTubeLength,3)}%
31 \PstLaboTestTube@EscapeTube@AdjustmentsStopper}}
32
33 % Macro for triple bent escape tube
34 \def\PstLaboTestTube@EscapeTube@triplebent{(%
35 \pssetlength{\pst@dimc}{\PstLaboTestTube@HeightTop}%
36 \psaddtolength{\pst@dimc}{\tw@}%
37 \pssetlength{\pst@dimd}{\PstLaboTestTube@EscapeTubeLength}%
38 \psaddtolength{\pst@dimd}{\@ne}%
39 \psline[doubleline=true,doublesep=0.1,linearc=0.5]
40   (0,0.5)(0,\pst@dimc)
41   (\PstLaboTestTube@EscapeTubeLength,\pst@dimc)
42   (\PstLaboTestTube@EscapeTubeLength,0)
43   (\pst@dimd,0)(\pst@dimd,1)}%
44 \PstLaboTestTube@EscapeTube@AdjustmentsStopper}}
45
46 % If there is no stopper, redraw of the external neck
47 \def\PstLaboTestTube@EscapeTube@AdjustmentsStopper{%
48 \ifPstLaboTestTube@Stopper
49 \else
50   \psellipticarc[linewidth=0.08](0,\PstLaboTestTube@HeightTop)
51     (\PstLaboTestTube@WidthNeckHalf,0.1){180}{0}
52 \fi}

```

```

53 \def\PstLaboTestTube@Stopper{%
54 \rput(0,\PstLaboTestTube@HeightTop){%
55   \pscustom[fillstyle=solid,fillcolor=lightgray]{%
56     \psellipticarc(0,0.5)(0.6,0.1){0}{180}
57     \psline(-0.6,0.5)(-0.4,-0.3)
58     \psellipticarc(0,-0.3)(0.4,0.1){180}{0}
59     \psline(0.4,-0.3)(0.6,0.5)}
60   \psellipticarc(0,0.5)(0.6,0.1){180}{0}
61   \psellipticarc[linewidth=0.08]
62     (\PstLaboTestTube@WidthNeckHalf,0.1){180}{0}
63 \ifPstLaboTestTube@EscapeTube\relax
64 \else
65   \let\psfillcolor\psdoublecolor
66   \pscustom[fillstyle=solid]{%
67     \psline[linearc=0.05](-0.07,0.7)(-0.07,0.5)(0.07,0.5)(0.07,0.7)}
68 \fi}
69
70
71 \def\PstLaboTestTube@ii(#1){%
72 \setkeys{psset}{dimen=middle}%
73 \use@par% Assignment of local parameters
74 \rput(#1){%
75   \rput(0,\PstLaboTestTube@HeightBottom){%
76     \expandafter\PstLaboTestTube@Tube
77     \expandafter\PstLaboTestTube@EscapeTube
78     \ifPstLaboTestTube@Stopper
79       \PstLaboTestTube@Stopper
80     \fi}}}}
81
82 \PstLaboTestTube[Form=roundbottom,EscapeTube=straight](-5,0)
83 \PstLaboTestTube[Stopper=true,EscapeTube=straight,
84   EscapeTubeLength=4](-2.5,0)
85 \PstLaboTestTube[unit=0.5,Stopper=true,EscapeTube=bent](-1,0)
86 \PstLaboTestTube[Form=small,Stopper=true,
87   EscapeTube=triplebent](2,0)

```



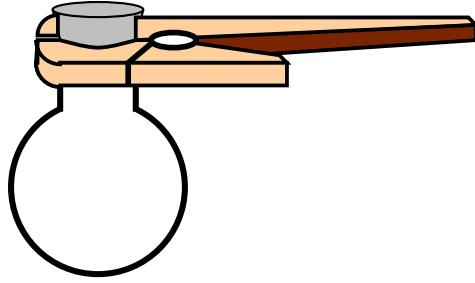
We can still easily add various objects to enrich the possibilities.

```

1 % Clip parameter
2 \newif\ifPstLaboTestTube@Clip
3 \define@key{psset}{Clip}[true]{\@nameuse{PstLaboTestTube@Clip#1}}
4
5 % Macro for clip
6 \def\PstLaboTestTube@Clip{%
7 % We put the clip at vertical position 0.7*height
8 \pst@dimh=\PstLaboTestTube@HeightTop\psyunit
9 \rput(-0.8,0.72\pst@dimh){%
10   \psset{linewidth=0.05}%
11   \pspolygon[fillstyle=solid,fillcolor=LightOrange]
12     (1.3,0.9)(5.7,0.9)(5.8,0.8)(1.8,0.6)(1.3,0.6)
13   \pspolygon[fillstyle=solid,fillcolor=Brown]
14     (1.3,0.6)(5.8,0.8)(5.8,0.6)(2.8,0.4)(1.8,0.6)
15   \pscustom[fillstyle=solid,fillcolor=LightOrange]{%
16     \pscurve(1.3,0.6)(0.8,0.5)(0.3,0.6)
17     \psline(0.3,0.6)(0,0.6)
18     \psellipticarc(0.3,0.3)(0.32,0.32){180}{270}
19     \psline(0.3,0.3)(3.3,0.3)(3.2,0.4)(1.8,0.6)(1.3,0.6)}
20   \pscustom[fillstyle=solid,fillcolor=LightOrange]{%
21     \psline(0,0.6)(0,0.3)
22     \psellipticarc(0.3,0.3)(0.32,0.32){180}{270}
23     \psline(0.3,0)(3.3,0)(3.3,0.3)(0.3,0.3)
24     \psellipticarcn(0.3,0.6)(0.32,0.32){-90}{180}}
25   \pswedge[fillstyle=solid,fillcolor=LightOrange]
26     (0.27,0.66){0.27}{90}{180}
27   \psellipse[linewidth=1.5\pslinewidth,fillstyle=solid]
28     (1.8,0.6)(0.3,0.1)
29   \psline[linewidth=1.5\pslinewidth](1.5,0.55)(1.2,0.3)(1.2,0)}}

30
31 \def\PstLaboTestTube@ii(#1){%
32   \setkeys{psset}{dimen=middle}%
33   \use@par% Assignment of local parameters
34   \rput(#1){%
35     \rput(0,\PstLaboTestTube@HeightBottom){%
36       \expandafter\PstLaboTestTube@Tube
37       \expandafter\PstLaboTestTube@EscapeTube
38       \ifPstLaboTestTube@Stopper
39         \PstLaboTestTube@Stopper
40       \fi
41       \ifPstLaboTestTube@Clip
42         \PstLaboTestTube@Clip
43       \fi}}}
44
45 \PstLaboTestTube[Form=roundbottom,Stopper=true,Clip=true]

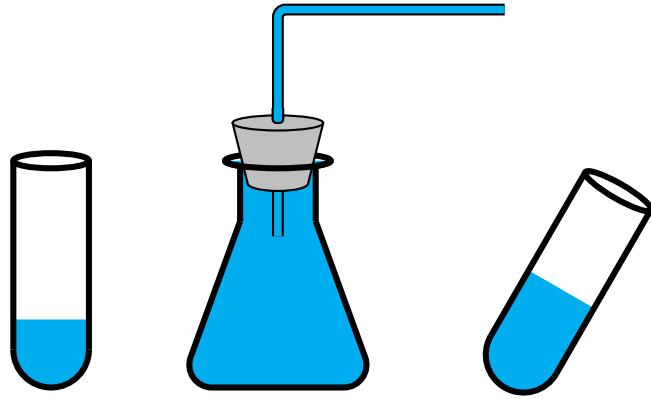
```



Then we may want to add a liquid inside the tube. Its level will be specified by real number between 0 and 1.

```

1 \SpecialCoor
2
3 % LiquidLevel parameter
4 \define@key{psset}{LiquidLevel}{%
5 \pst@checknum{\#1}{\PstLaboTestTube@LiquidLevel}}
6
7 \setkeys{psset}{LiquidLevel=0}% Default value
8
9 % Macro for liquid
10 \def\PstLaboTestTube@Liquid#1{%
11 \psframe[linestyle=none,fillstyle=solid,fillcolor=cyan]
12 (-4,-2)(! 4 \PstLaboTestTube@Height\space #1 mul)}
13
14 \def\PstLaboTestTube@ii(#1){%
15 \setkeys{psset}{dimen=middle}%
16 \use@par% Assignment of local parameters
17 \rput(#1){%
18 \psclip{\pscustom[linestyle=none]{%
19 \expandafter\PstLaboTestTube@Tube}}
20 \PstLaboTestTube@Liquid{\PstLaboTestTube@LiquidLevel}
21 \endpsclip
22 \rput(0,\PstLaboTestTube@HeightBottom){%
23 % ...
24 \fi}}}}
25
26 \PstLaboTestTube[LiquidLevel=0.3](-3,0)
27 \PstLaboTestTube[Form=erlenmeyer,Stopper=true,
28 EscapeTube=bent,LiquidLevel=1,doublecolor=cyan]
29 \rput{-30}(3,0){\PstLaboTestTube[LiquidLevel=0.5]}
```



Nevertheless, we can see in the last previous example that the position of the liquid is not physically significant if the tube is rotated. This is a difficult geometric problem, for which we give here an approximative solution, sufficient for our needs.

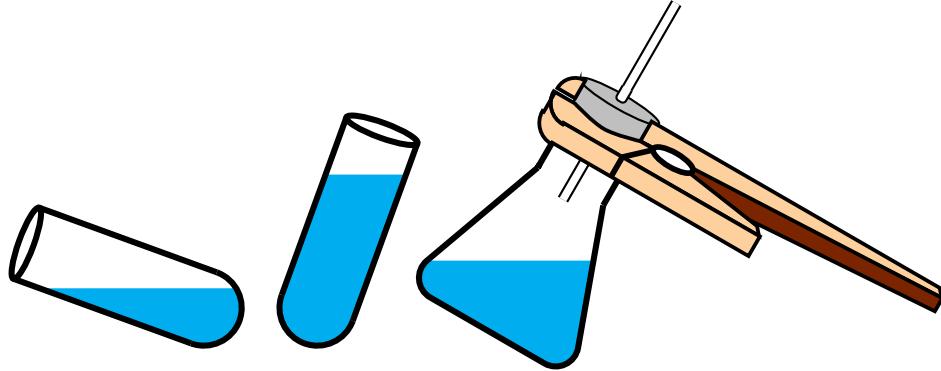
```

1 % Angle parameter
2 \define@key{psset}{Angle}{%
3   \pst@getangle{\#1}\PstLaboTestTube@Angle}
4
5 \setkeys{psset}{Angle=0}% Default value
6
7 % Macro for liquid (#1 = angle, #2 = liquid level)
8 \def\PstLaboTestTube@Liquid#1#2{%
9   \pst@Verb{%
10     /xA \PstLaboTestTube@WidthBottomHalf\space
11       #1 0 gt {neg} if} def
12     /LiquidHeight {\#2 \PstLaboTestTube@Height\space mul
13       #2 \PstLaboTestTube@HeightBottom\space mul
14       #1 sin abs mul sub} def
15   /xA' {xA #1 cos mul
16     \PstLaboTestTube@Height\space #1 sin mul sub} def
17   #1 0 gt {/Xinf xA' def /Xsup 2 def}
18     {/Xinf -2 def /Xsup xA' def} ifelse}%
19 \psframe[linestyle=none,fillstyle=solid,fillcolor=cyan]
20   (! Xinf -2)(! Xsup LiquidHeight)}
21
22 \def\PstLaboTestTube@ii(#1){%
23   \setkeys{psset}{dimen=middle}%
24   \use@par% Assignment of local parameters
25   \rput(#1){%
26     \psclip{\rput{\PstLaboTestTube@Angle}
27       (0,\PstLaboTestTube@HeightBottom){%
28         \pscustom[linestyle=none]{%
29           \translate(0,-\PstLaboTestTube@HeightBottom)
30           \expandafter\PstLaboTestTube@Tube}}}
31       \PstLaboTestTube@Liquid{\PstLaboTestTube@Angle}
32         {\PstLaboTestTube@LiquidLevel}%
33   \endpsclip
34   \rput{\PstLaboTestTube@Angle}
```

```

35   (0,\PstLaboTestTube@HeightBottom){%
36     \expandafter\PstLaboTestTube@Tube
37   % ...
38   \fi}}}}}
39
40 \PstLaboTestTube[Angle=70,LiquidLevel=0.3](-4,0)
41 \PstLaboTestTube[Angle=-20,LiquidLevel=0.8](-2.5,0)
42 \PstLaboTestTube[Angle=-30,Form=erlenmeyer,Stopper=true,
43   Clip=true,EscapeTube=straight,LiquidLevel=0.5](0.5,0)

```



We can now parameterize the shading of the liquid. For this, we only have to parameterize the style of the polygon drawn to show the liquid.

```

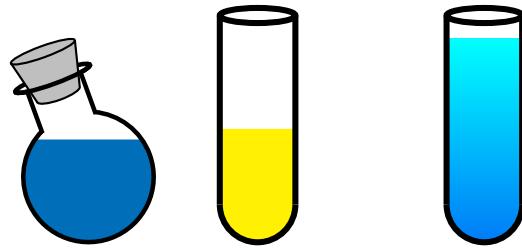
1  % LiquidType parameter
2  \define@key{psset}{LiquidType}{%
3    \def\PstLaboTestTube@LiquidType{\#1}}
4
5  % Default value (first is the parameter name,
6  % second is the aspect name!)
7  \setkeys{psset}{LiquidType=LiquidType}
8  \newpsstyle{LiquidType}{%
9    linestyle=none,fillstyle=solid,fillcolor=cyan}
10
11 % Macro for liquid (#1 = angle, #2 = liquid level, #3 = style)
12 \def\PstLaboTestTube@Liquid#1#2#3{%
13   \pst@Verb{%
14     ...
15     {/Xinf -2 def /Xsup xA' def} ifelse}%
16   \psframe[style=#3](! Xinf -2) (! Xsup LiquidHeight)}
17
18 \def\PstLaboTestTube@ii(#1){%
19   \setkeys{psset}{dimen=middle}%
20   \use@par% Assignment of local parameters
21   \rput(#1){%
22     \psclip{\rput{\PstLaboTestTube@Angle}
23       (0,\PstLaboTestTube@HeightBottom){%
24         \pscustom*[linestyle=none]{%
25           \translate(0,-\PstLaboTestTube@HeightBottom)
26           \expandafter\PstLaboTestTube@Tube}}}
27     \PstLaboTestTube@Liquid{\PstLaboTestTube@Angle}

```

```

28   \endpsclip
29   % ...
30   \fi}}}}
31
32 \newpsstyle{cobalt}{%
33   linestyle=none,fillstyle=solid,fillcolor=NavyBlue}
34 \newpsstyle{diffusion}{%
35   linestyle=none,fillstyle=gradient,gradmidpoint=0}
36 \newpsstyle{oil}{linestyle=none,fillstyle=solid,fillcolor=yellow}
37
38 \PstLaboTestTube[unit=0.75,Form=roundbottom,Stopper=true,
39   Angle=20,LiquidType=cobalt,LiquidLevel=0.7](-3,0)
40 \PstLaboTestTube[LiquidType=oil,LiquidLevel=0.5]
41 \PstLaboTestTube[LiquidType=diffusion,LiquidLevel=0.9](3,0)

```



If we want to manage several liquids (here up to 3), we have only to loop on them. Note that we use a number inside the macro names. We could have used the `\@namedef` macro for that, but here we want to call the `\pst@checknum` macro to verify if the value is a well formatted number. `\pst@checknum` does not allow numeric characters inside macro names, so we use a variation on it, the `\@nameedef` macro, which does allow numeric characters inside macro names.

```

1  % LiquidLevel parameters
2  \define@key{psset}{LiquidLevel1}{%
3    \pst@checknum{#1}{\pst@tempb}%
4    \expandafter\edef\csname PstLaboTestTube@LiquidLevel1%
5      \endcsname{\pst@tempb}%
6  \define@key{psset}{LiquidLevel2}{%
7    \pst@checknum{#1}{\pst@tempb}%
8    \expandafter\edef\csname PstLaboTestTube@LiquidLevel2%
9      \endcsname{\pst@tempb}%
10 \define@key{psset}{LiquidLevel3}{%
11   \pst@checknum{#1}{\pst@tempb}%
12   \expandafter\edef\csname PstLaboTestTube@LiquidLevel3%
13     \endcsname{\pst@tempb}%
14
15 % LiquidType parameters
16 \define@key{psset}{LiquidType1}{%
17   \@namedef{PstLaboTestTube@LiquidType1}{#1}%
18 \define@key{psset}{LiquidType2}{%

```

```

19 \@namedef{PstLaboTestTube@LiquidType2}{#1}}
20 \define@key{psset}{LiquidType3}{%
21 \@namedef{PstLaboTestTube@LiquidType3}{#1}}
22
23 % Default values
24 \setkeys{psset}{LiquidType1=LiquidType1,LiquidLevel1=0,
25 LiquidType2=,LiquidLevel2=0,LiquidType3=,LiquidLevel3=0}
26
27 % Default style
28 \newpsstyle{LiquidType1}{%
29   linestyle=none,fillstyle=solid,fillcolor=cyan}
30
31 \def\PstLaboTestTube@ii(#1){%
32 \setkeys{psset}{dimen=middle}%
33 \use@par% Assignment of local parameters
34 \rput(#1){%
35   \psclip{\rput{\PstLaboTestTube@Angle}
36     (0,\PstLaboTestTube@HeightBottom)}{%
37     \pscustom*[linestyle=none]{%
38       \translate(0,-\PstLaboTestTube@HeightBottom)
39       \expandafter\PstLaboTestTube@Tube}}}
40   \multido{\iLiquid=\@ne+@\ne}{\thr@ @}{%
41     \ifdim\@nameuse{PstLaboTestTube@LiquidLeve\iLiquid}p@>\z@
42     \PstLaboTestTube@Liquid{\PstLaboTestTube@Angle}
43     {\@nameuse{PstLaboTestTube@LiquidLeve\iLiquid}}
44     {\@nameuse{PstLaboTestTube@LiquidType\iLiquid}}
45   }
46   \endpsclip
47   % ...
48   \fi}}}}
49
50 \newpsstyle{vinegar}{linestyle=none,fillstyle=solid,fillcolor=magenta}
51
52 \PstLaboTestTube[Form=roundbottom,Stopper=true,
53   LiquidType1=oil,LiquidLevel1=0.6,
54   LiquidType2=vinegar,LiquidLevel2=0.2](-3,0)
55 \PstLaboTestTube[Angle=-20,LiquidType1=cobalt,LiquidLevel1=0.8,
56   LiquidType2=oil,LiquidLevel2=0.7,
57   LiquidType3=vinegar,LiquidLevel3=0.5](3,0)

```



We may want to change the form of the liquid surface. This is easy to do.

```

1 % Liquid surface parameter
2 \define@key{psset}{LiquidSurface}{%
3 \def\PstLaboTestTube@LiquidSurface{%
4 \@nameuse{PstLaboTestTube@LiquidSurface@#1}}}
5
6 % Default value
7 \setkeys{psset}{LiquidSurface=straight}
8
9 % Macro for liquid surface (straight)
10 % (#1 = angle, #2 = liquid level, #3 = style)
11 \def\PstLaboTestTube@LiquidSurface@straight#1#2#3{%
12 \pst@Verb{%
13 % ...
14         {/Xinf -2 def /Xsup xA' def} ifelse}%
15 \psframe[style=#3](! Xinf -2) (! Xsup LiquidHeight)}
16
17 % Macro for liquid surface (wavy)
18 % (#1 = angle, #2 = liquid level, #3 = style)
19 \def\PstLaboTestTube@LiquidSurface@wavy#1#2#3{%
20 \pst@Verb{%
21 % ...
22         {/Xinf -2 def /Xsup xA' def} ifelse}%
23 \pscustom[style=#3]{%
24 \psline(! Xinf LiquidHeight) (! Xinf -2) (! Xsup -2) (! Xsup LiquidHeight)
25 \pszigzag [linearc=0.1,coilwidth=0.2,coilheight=2.5]
26         (! Xinf LiquidHeight)}}
27
28 \def\PstLaboTestTube@ii(#1){%
29 \setkeys{psset}{dimen=middle}%
30 \use@par% Assignment of local parameters
31 \rput(#1){%
32     \psclip{\rput{\PstLaboTestTube@Angle}
33             (0,\PstLaboTestTube@HeightBottom){%
34                 \pscustom [linestyle=none]{%
35                     \translate(0,-\PstLaboTestTube@HeightBottom)
36                     \expandafter\PstLaboTestTube@Tube}}}
37     \multido{\iLiquid=\@ne+\@ne}{\thr@@}{%
38         \ifdim\@nameuse{PstLaboTestTube@LiquidLevel\iLiquid}\p@>\z@
39             \expandafter\PstLaboTestTube@LiquidSurface%
40             {\PstLaboTestTube@Angle}
41             {\@nameuse{PstLaboTestTube@LiquidLevel\iLiquid}}
42             {\@nameuse{PstLaboTestTube@LiquidType\iLiquid}}}
43     \fi}
44     \endpsclip
45 % ...
46 \fi}}}}
47
48 \psset{LiquidSurface=wavy}
49 \PstLaboTestTube[Form=small,Stopper=true,
50     LiquidType1=cobalt,LiquidLevel1=0.3](-3,0)
51 \PstLaboTestTube[Angle=-20,LiquidType1=cobalt,LiquidLevel1=0.8,
52     LiquidType2=oil,LiquidLevel2=0.6,
```

53 | LiquidType3=vinegar,LiquidLevel3=0.5](3,0)



We can also add various substances inside the tubes. Bubbles and nails are defined below, in random sizes and at random positions.

```
1 % Substance parameter
2 \define@key{psset}{Substance}{%
3 \def\PstLaboTestTube@Substance{%
4 \@nameuse{PstLaboTestTube@Substance@#1}}}
5
6 % Substance number parameter
7 \define@key{psset}{SubstanceNumber}{%
8 \pst@getint{#1}{\PstLaboTestTube@SubstanceNumber}}
9
10 % Substance style parameter
11 \define@key{psset}{SubstanceStyle}{%
12 \edef\PstLaboTestTube@SubstanceStyle{#1}}
13
14 % Default values
15 \setkeys{psset}{Substance=,SubstanceNumber=10,SubstanceStyle=}
16
17 % Macro to draw bubbles of random sizes at random positions
18 \def\PstLaboTestTube@Substance@bubbles{%
19 \PstLaboTestTube@Substance@i{%
20 \psdot[dotstyle=o,dotsize=\pointless\pst@dimh]
21 (! \pst@number{\pst@dimc} \pst@number{\pst@dimd}
22 @nameuse{PstLaboTestTube@LiquidLevel1} mul)})}
23
24 % Macro to draw colored bubbles of random sizes at random pos
25 \def\PstLaboTestTube@Substance@coloredbubbles{%
26 \PstLaboTestTube@Substance@i{%
27 \pscircle[linewidth=0.5\pslinewidth,fillstyle=ccslope,
28 runit=\pointless\pst@dimh]
29 (! \pst@number{\pst@dimc} \pst@number{\pst@dimd}
30 @nameuse{PstLaboTestTube@LiquidLevel1} mul){0.06}}}
31
32 % Macro to draw nails of random sizes at random positions
33 \def\PstLaboTestTube@Substance@nails{%
34 \PstLaboTestTube@Substance@i{%
35 \rput{\the\pst@cnth}{%
36 (! \pst@number{\pst@dimc} \pst@number{\pst@dimd}
37 @nameuse{PstLaboTestTube@LiquidLevel1} mul){%
38 \psset{unit=\pointless\pst@dimh}}}}
```

```

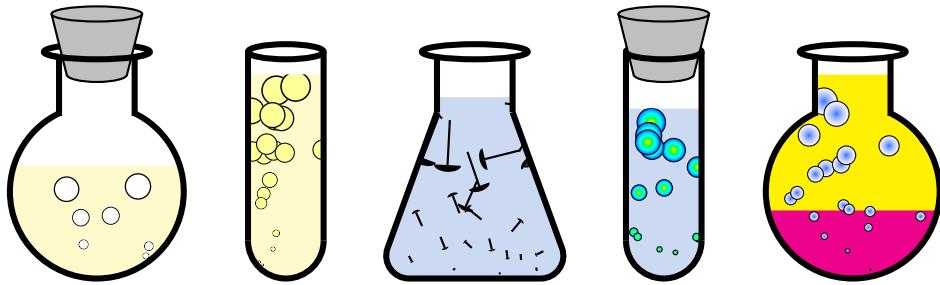
39   \psline(0,0.2)
40   \psellipticarc*(0,0.2)(0.06,0.03){0}{180}}}}
41
42 % Macro to draw thin objects of random sizes at random pos.
43 \def\PstLaboTestTube@Substance@i#1{%
44   #1 = macro of the object
45   \multido{\iSubstance=\@ne+\@ne}
46     {\PstLaboTestTube@SubstanceNumber}%
47     \setrandim{\pst@dimc}{\PstLaboTestTube@WidthBottomHalf\p@}
48       {\PstLaboTestTube@WidthBottomHalf\p@}% For position
49   % For position and size
50   \setrandim{\pst@dimd}{\z@}{\PstLaboTestTube@Height\p@}%
51   \setrnum{\pst@cnth}{\z@}{360}% For angle
52   % The size increase proportionally to the vertical position
53   \pst@dimh=\pst@dimd
54   % And this size must depend of the unit
55   \pst@dimg=0.04\psunit
56   \pst@dimh=\pointless\pst@dimg\pst@dimh
57   \ifx\PstLaboTestTube@SubstanceStyle\@empty
58     \else
59       \setkeys{psset}{style=\PstLaboTestTube@SubstanceStyle}%
60   \fi
61   #1}}
62
63 \def\PstLaboTestTube@ii(#1){%
64   \setkeys{psset}{dimen=middle}%
65   \use@par% Assignment of local parameters
66   \rput(#1){%
67     \psclip{\rput{\PstLaboTestTube@Angle}
68       (0,\PstLaboTestTube@HeightBottom)}{%
69       \pscstyle[linestyle=none]{%
70         \translate(0,-\PstLaboTestTube@HeightBottom)
71         \expandafter\PstLaboTestTube@Tube}}}
72     % To clip the optional substance by the first liquid
73     \expandafter\PstLaboTestTube@LiquidSurface%
74       {\PstLaboTestTube@Angle}
75       {\@nameuse{\PstLaboTestTube@LiquidLevel1}{none}}}
76     % ...
77     \PstLaboTestTube@Substance
78   \endpsclip
79   % ...
80   \vfi}}}}
81
82 \newpsstyle{champagne}{%
83   linestyle=none,fillstyle=solid,fillcolor=LemonChiffon}
84 \newpsstyle{water}{%
85   linestyle=none,fillstyle=solid,fillcolor=LightBlue}
86 \newpsstyle{BubblesChampagne}{%
87   fillstyle=solid,fillcolor=PaleYellow}
88
89 \PstLaboTestTube[Form=roundbottom,Stopper=true,
90   LiquidType1=champagne,LiquidLevel1=0.5,
91   Substance=bubbles]{-5,0}
92 \PstLaboTestTube[LiquidType1=champagne,

```

```

92 | LiquidLevel1=0.9,Substance=bubbles,SubstanceNumber=20,
93 | SubstanceStyle=BubblesChampagne](-2.5,0)
94 | \PstLaboTestTube[Form=erlenmeyer,LiquidType1=water,
95 | LiquidLevel1=0.8,Substance=nails,SubstanceNumber=30]
96 | \PstLaboTestTube[Stopper=true,LiquidType1=water,LiquidLevel1=0.75,
97 | Substance=coloredbubbles,SubstanceNumber=15](2.5,0)
98 | \PstLaboTestTube[Form=roundbottom,LiquidType1=oil,
99 | LiquidLevel1=0.9,LiquidType2=vinegar,LiquidLevel2=0.3,
100 | Substance=coloredbubbles,SubstanceNumber=25,
101 | slopebegin=NavyBlue,slopeend=white](5,0)

```



And we can of course compose several objects to illustrate complex scenes.

```

1 % Macro for becher
2 \def\PstLaboTestTube@becher{%
3 \gdef\PstLaboTestTube@Height{2.5}%
4 \gdef\PstLaboTestTube@HeightTop{1.275}%
5 \gdef\PstLaboTestTube@WidthBottomHalf{0.55}%
6 \gdef\PstLaboTestTube@WidthTopHalf{0.55}%
7 \gdef\PstLaboTestTube@WidthNeckHalf{0.55}%
8 \psset{dimen=middle,linewidth=0.08,%
9 origin={0,\PstLaboTestTube@becher@HeightBottom}}%
10 \psellipse(0,2.55)(1.1,0.1)
11 \psline[linearc=0.5](-1,2.5)(-1,0)(1,0)(1,2.5)
12 \multido{\nDiv=0.3+0.5}{4}{%
13 \psline[linewidth=0.5\pslinewidth](-0.75,\nDiv)(-0.3,\nDiv)}
14 \multido{\nSubDiv=0.3+0.1}{16}{%
15 \psline[linewidth=0.25\pslinewidth](-0.75,\nSubDiv)(-0.5,\nSubDiv)}}
16 \def\PstLaboTestTube@becher@HeightBottom{1.275}
17
18 % Macro for "Bunsen burner" object
19 \def\PstLaboBunsenBurner{%
20 \def\pst@par{} \pst@object{PstLaboBunsenBurner}}
21
22 \def\PstLaboBunsenBurner@i{%
23 % Test for optional angle (default = 0)
24 \@ifnextchar\bgroup{\PstLaboBunsenBurner@ii}{%
25 {\PstLaboBunsenBurner@ii{#1}}}}
26
27 \def\PstLaboBunsenBurner@ii#1{%
28 #1 = angle
29 % Test for optional coordinate (default = (0,0))
30 \@ifnextchar({\PstLaboBunsenBurner@iii}{#1}}}

```

```

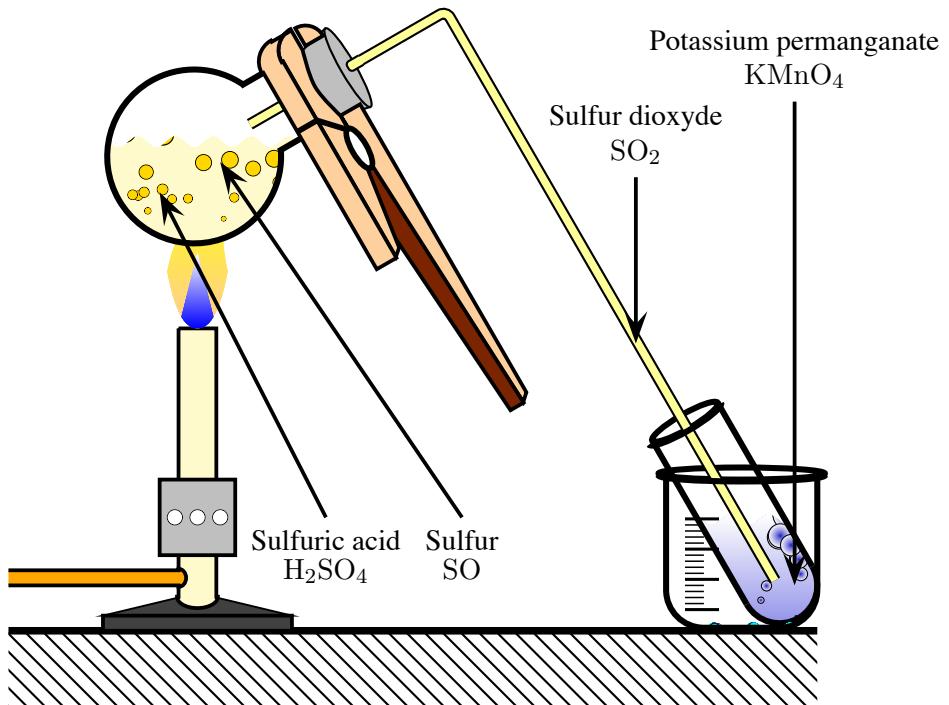
30          {\PstLaboBunsenBurner@iii{\#1}{\z@,\z@})}
31
32 \def\PstLaboBunsenBurner@iii#1#2{{%
33 % #1 = angle, #2 = coordinate
34 \setkeys{psset}{linewidth=0.05}%
35 \use@par% Assignment of local parameters
36 \rput{#1}{#2}{%
37     % Basis
38     \pstriangle[fillstyle=solid,fillcolor=darkgray,linewidth=0.5\pslinewidth]
39         (0,0.2)(2.5,0.3)
40     \psframe[fillstyle=solid,fillcolor=darkgray](-1.25,0)(1.25,0.2)
41     % Burner
42     \pscustom[fillstyle=solid,fillcolor=LemonChiffon]{%
43         \psline(-0.25,4)(-0.25,0.45)
44         \psellipticarc(0,0.45)(0.27,0.15){180}{0}
45         \psline(0.25,0.45)(0.25,4)(-0.25,4)}
46     % Regulating flame
47     \psframe[fillstyle=solid,fillcolor=lightgray](-0.5,1)(0.5,2)
48     \psdots[dotstyle=0,dotscale=1.25](-0.3,1.5)(0,1.5)(0.3,1.5)
49     % Small gaz tube
50     \pscustom[fillstyle=solid,fillcolor=Orange]{%
51         \psline(-2.5,0.6)(-0.25,0.6)
52         \psellipticarc(-0.25,0.7)(0.1,0.12){-90}{90}
53         \psline(-0.25,0.8)(-2.5,0.8)}
54     % Flame
55     \rput(0,4){%
56         \psset{linestyle=none,fillstyle=gradient,gradmidpoint=0}%
57         \psclip{\psbezier[gradbegin=LightOrange,gradend=yellow]%
58             (-0.25,0)(-0.35,0.5)(-0.4,0.75)
59             (-0.35,1)(-0.25,1.5)(0.5,2)
60             (0.25,1.5)(0.35,1)(0.4,0.75)
61             (0.35,0.5)(0.25,0)(0,0)}
62         \pspolygon[gradbegin=blue,gradend=white](-0.25,0)(0.25,0)(0,1)
63         \endpsclip}}}}
64
65 \newpsstyle{SulphuricAcid}{linestyle=none,fillstyle=solid,
66     fillcolor=LemonChiffon}
67
68 \newpsstyle{PotassiumPermanganate}{linestyle=none,fillstyle=gradient,
69     gradmidpoint=1,gradbegin=white,gradend=DarkViolet}
70
71 \newpsstyle{SulphuricDioxyde}{fillstyle=solid,fillcolor=Gold}
72
73 \psset{Substance=coloredbubbles,LiquidSurface=wavy}%
74 % Table
75 \psline[linewidth=0.1](-5,0)(5.7,0)
76 \psframe[linestyle=none,fillstyle=vlines](-5,0)(5.7,-1)
77 % Bunsen burner
78 \PstLaboBunsenBurner(-2.5,0)
79 % Tube on right
80 \PstLaboTestTube[Angle=30,
81     LiquidType1=PotassiumPermanganate,LiquidLevel1=0.5,
82     Substance=coloredbubbles,

```

```

83 | slopebegin=DarkViolet,slopeend=white](5.2,0.1)%
84 | % Becher
85 | \PstLaboTestTube[yunit=0.8,Form=becher](4.7,0.05)%
86 | % Round bottom flask on left
87 | \PstLaboTestTube[Form=roundbottom,Angle=-60,
88 | Stopper=true,Clip=true,
89 | EscapeTube=bent,EscapeTubeLength=8.7,doublecolor=PaleYellow,
90 | LiquidType1=SulphuricAcid,LiquidLevel1=0.7,
91 | Substance=bubbles,SubstanceNumber=30,
92 | SubstanceStyle=SulphuricDioxyde](-2.5,5.1)%
93 | % Adjustment to redraw the external neck of the right tube
94 | \rput{30}(5.45,0.14){%
95 |   \psellipticarc[lineWidth=0.08](0,3)(0.55,0.1){180}{0}%
96 | % Adjustment to redraw the external neck of the becher
97 | \rput(4.7,2.055){%
98 |   \psellipticarc[yunit=0.8,lineWidth=0.08](1.1,0.1){180}{0}%
99 | % Labels
100 | \psset{lineWidth=0.05,arrows=->,arrowscale=2}%
101 | \psline(-0.8,1.5)(-3,5.8)
102 |   \uput[-90]{-0.8,1.5}{%
103 |     \shortstack{Sulfuric acid\\$\\mathrm{H_2SO_4}$}}
104 | \psline(1,1.5)(-2.2,6.2)
105 |   \uput[-90]{1,1.5}{\shortstack{Sulfur\\$\\mathrm{SO}$}}
106 | \psline(3.3,6)(3.3,3.8)
107 |   \uput[90]{3.3,6}{\shortstack{Sulfur dioxyde\\$\\mathrm{SO_2}$}}
108 | \psline(5.4,7)(5.4,0.6)
109 |   \uput[90]{5.4,7}{%
110 |     \shortstack{Potassium permanganate\\$\\mathrm{KMnO_4}$}}

```



64.3 Advanced example: Gantt charts

This is another complex example which builds high-level tools to draw simple Gantt charts. We will define a macro (`\PstGanttTask`) to describe tasks and will show how to define an environment (`PstGanttChart`) to manage all these defined tasks. Both must of course accept optional parameters.

```
1 % Parameters definition
2 % =====
3
4 % Intervals to show?
5 \newif\ifPstGantt@ChartShowIntervals
6 \define@key{psset}{ChartShowIntervals}[true]{%
7 \@nameuse{PstGantt@ChartShowIntervals#1}}
8
9 % Style for the tasks
10 \define@key{psset}{TaskStyle}{\edef\PstGantt@TaskStyle{\#1}}
11
12 % Name for unit interval
13 \define@key{psset}{ChartUnitIntervalName}{%
14 \edef\PstGantt@ChartUnitIntervalName{\#1}}
15
16 % Name for basic unit
17 \define@key{psset}{ChartUnitBasicIntervalName}{%
18 \edef\PstGantt@ChartUnitBasicIntervalName{\#1}}
19
20 % Unit interval for the tasks
21 % (7 for a week, 30 for a month, etc.)
22 % Warning: define it before "TaskUnitType"!
23 \define@key{psset}{TaskUnitIntervalValue}{%
24 \pst@getint{\#1}\PstGantt@TaskUnitIntervalValue}
25
26 % Unit type for the tasks
27 % ("UnitIntervalName" or "UnitBasicIntervalName")
28 \define@key{psset}{TaskUnitType}{%
29 \edef\PstGantt@TaskUnitValue{\#1}%
30 % Validation of the parameter
31 \ifx\PstGantt@TaskUnitValue\PstGantt@ChartUnitIntervalName
32     \edef\PstGantt@TaskUnitValue{%
33         \PstGantt@TaskUnitIntervalValue}%
34 \else
35     \ifx\PstGantt@TaskUnitValue%
36         \PstGantt@ChartUnitBasicIntervalName
37     \def\PstGantt@TaskUnitValue{1}%
38 \else
39     {\@pstrickserr{GanttTaskUnitType must be
40         '\PstGantt@ChartUnitIntervalName'}
```

```

41      or '\PstGantt@ChartUnitBasicIntervalName'
42      (and not '\PstGantt@TaskUnitValue')}\@eha}%
43      \fi
44 \fi}

45
46 % Outside label for the tasks
47 \define@key{psset}{TaskOutsideLabel}{%
48 \def\PstGantt@TaskOutsideLabel{\#1}%
49
50 % Inside label for the tasks
51 \define@key{psset}{TaskInsideLabel}{%
52 \def\PstGantt@TaskInsideLabel{\#1}%
53
54 % Maximum outside size label for the tasks
55 % (in unit "TaskUnitType" !)
56 \define@key{psset}{TaskOutsideLabelMaxSize}{%
57 \pst@getint{\#1}\PstGantt@TaskOutsideLabelMaxSize}%
58
59 % Default values
60 % =====
61 % Don't show intervals, default task style, unit for tasks
62 % is a week (so 7 days), no outside and inside labels
63 \setkeys{psset}{%
64 ChartShowIntervals=false,TaskStyle=TaskStyleDefault,
65 ChartUnitIntervalName=Week,ChartUnitBasicIntervalName=Day,
66 TaskUnitIntervalValue=7,TaskUnitType=Week,TaskOutsideLabel=,
67 TaskInsideLabel=,TaskOutsideLabelMaxSize=0}%
68
69 % Task default style is yellow background
70 \newpsstyle{TaskStyleDefault}{fillstyle=solid,fillcolor=yellow}%
71
72 % The environment PstGanttChart
73 % =====
74
75 % Syntax:\PstGanttChart[parameters]{Nb of tasks}{Nb of days}
76 %           \endPstGanttChart
77 % or
78 % \begin{PstGanttChart}[parameters]{Nb of tasks}{Nb of days}
79 % \end{PstGanttChart}
80 \def\PstGanttChart{\def\pst@par{}\pst@object{PstGanttChart}}%
81
82 \def\PstGanttChart@#1#2{%
83 \bgroup
84   \setkeys{psset}{unit=0.1}%
85   \use@par% Assignment of local parameters
86   %

```

```

87  % "pspicture" environment
88  \pst@cnta=\PstGantt@TaskOutsideLabelMaxSize
89  \multiply\pst@cnta\PstGantt@TaskUnitValue
90  %
91  \pst@cntb=#1
92  \multiply\pst@cntb by 5
93  \advance\pst@cntb\@ne
94  %
95  \pst@cntc=#2
96  \multiply\pst@cntc\PstGantt@TaskUnitValue
97  \advance\pst@cntc\@tw@
98  %
99  \ifPstGantt@ChartShowIntervals
100   \pspicture(-\pst@cnta,-\pst@cntb)(\pst@cntc,\@two)
101 \else
102   \pspicture(-\pst@cnta,-\pst@cntb)(\pst@cntc,\z@)
103 \fi
104 \psframe(\z@,-\pst@cntb)(\pst@cntc,\z@)
105 %
106 \ifPstGantt@ChartShowIntervals
107   % We will show the intervals
108   \pst@cnta=#2
109   \multiply\pst@cnta\PstGantt@TaskUnitValue
110   \divide\pst@cnta\PstGantt@TaskUnitIntervalValue
111   \advance\pst@cnta\@ne
112   %
113   \pst@cntb=#1
114   \multiply\pst@cntb by 5
115   \advance\pst@cntb\@ne
116   %
117   \pst@dima=\PstGantt@TaskUnitIntervalValue\p@
118   \divide\pst@dima\@tw@
119   \advance\pst@dima\@ne\p@
120   %
121   \multido{\iInterval=\@ne+\@ne,
122         \iIntervalPos=\@ne+
123         \PstGantt@TaskUnitIntervalValue,
124         \rIntervalPos=\pst@number{\pst@dima}+%
125         \PstGantt@TaskUnitIntervalValue}{\pst@cnta}%
126   \ifnum\iInterval=\pst@cnta
127     \psline(\iIntervalPos,\z@)(\iIntervalPos,1.5)
128     \psline[linestyle=dotted](\iIntervalPos,-\pst@cntb)
129                               (\iIntervalPos,\z@)
130   \else
131     \rput(\rIntervalPos,\@ne){%
132       \PstGantt@ChartUnitIntervalName{} \iInterval}

```

```

133          \psline(\iIntervalPos,\z@)(\iIntervalPos,1.5)
134          \psline[linestyle=dotted](\iIntervalPos,-\pst@cntb)
135                                (\iIntervalPos,\z@)
136      \fi}
137  \fi}
138
139 \def\endPstGanttChart{%
140   \endpspicture % End of "pspicture" environment
141 \egroup}
142
143 % The macro \PstGanttTask
144 % =====
145
146 \newcount\PstGantt@TaskCnt
147 \PstGantt@TaskCnt=\z@
148
149 % Syntax: \PstGanttTask[parameters]{Start}{Length}
150 \def\PstGanttTask{\def\pst@par{}\pst@object{\PstGanttTask}}
151
152 \def\PstGanttTask@#1#2{%
153 \advance\PstGantt@TaskCnt\m@ne % To increment globally
154 \bgroup
155   \use@par% Assignment of local parameters
156   % Frame
157   \pst@cnta=\PstGantt@TaskUnitValue
158   \multiply\pst@cnta by #1
159   \advance\pst@cnta\@ne
160   %
161   \pst@cntb=\PstGantt@TaskUnitValue
162   \multiply\pst@cntb by #2
163   \advance\pst@cntb\pst@cnta
164   %
165   \pst@cntc=\PstGantt@TaskCnt
166   \multiply\pst@cntc by 5
167   %
168   \pst@cntd=\pst@cntc
169   \advance\pst@cntd by 4
170   %
171   \psframe[style=\PstGantt@TaskStyle](\pst@cnta,\pst@cntc)
172                               (\pst@cntb,\pst@cntd)
173   % Inside label
174   \ifx\PstGantt@TaskInsideLabel\@empty
175   \else
176     \pst@dima=\pst@cnta\p@
177     \advance\pst@dima\pst@cntb\p@
178     \divide\pst@dima\tw@

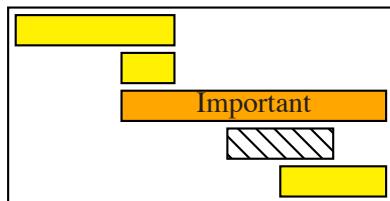
```

```

179 %
180 \pst@dimb=\pst@cntc\p@
181 \advance\pst@dimb\pst@cntd\p@
182 \divide\pst@dimb\tw@
183 %
184 \rput{\pst@number{\pst@dima},\pst@number{\pst@dimb}}{%
185   \PstGantt@TaskInsideLabel}
186 \fi
187 % Outside label
188 \ifx\PstGantt@TaskOutsideLabel\@empty
189 \else
190   \pst@dima=\pst@cntc\p@
191   \advance\pst@dima\pst@cntd\p@
192   \divide\pst@dima\tw@
193   \rput[r]{-1.5}{\pst@number{\pst@dima}}{%
194     \PstGantt@TaskOutsideLabel}
195 \fi
196 \egroup

```

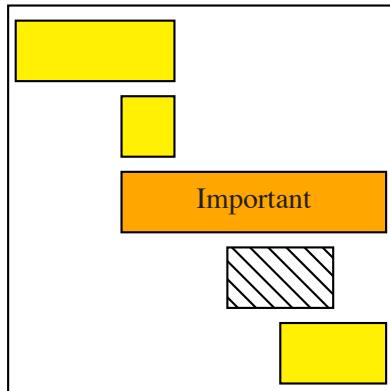
We can now use this new environment to specify Gantt charts, inserting into it as many tasks as we need (we use here the L^AT_EX flavor syntax, which of course must be adapted if PLAIN T_EX or ConT_EXt is used.)



```

1 \newpsstyle{Important}{fillstyle=solid,fillcolor=Orange}
2 \newpsstyle{NotImportant}{fillstyle=vlines}
3
4 \begin{PstGanttChart}{5}{7}
5   \PstGanttTask{0}{3}
6   \PstGanttTask{2}{1}
7   \PstGanttTask[TaskStyle=Important,
8                 TaskInsideLabel=Important]{2}{5}
9   \PstGanttTask[TaskStyle=NotImportant]{4}{2}
10  \PstGanttTask{5}{2}
11 \end{PstGanttChart}

```



```

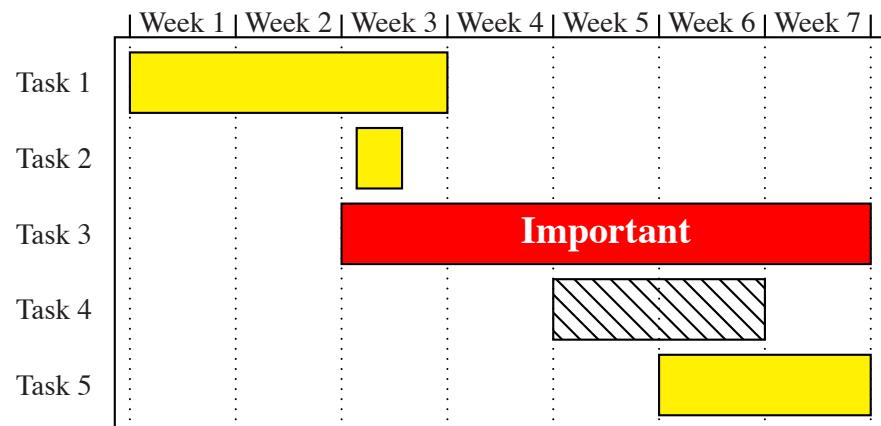
1 \begin{PstGanttChart}[yunit=2]{5}{7} % Same with double "yunit"
2   \PstGanttTask{0}{3}
3   \PstGanttTask{2}{1}
4   \PstGanttTask[TaskStyle=Important,
5                 TaskInsideLabel=Important]{2}{5}
6   \PstGanttTask[TaskStyle=NotImportant]{4}{2}
7   \PstGanttTask{5}{2}
8 \end{PstGanttChart}

```

```

1 \newpsstyle{Important}{fillstyle=solid,fillcolor=red}
2
3 \begin{PstGanttChart}[unit=2,TaskOutsideLabelMaxSize=1,
4   ChartShowIntervals=true]{5}{7}
5   \PstGanttTask[TaskOutsideLabel={Task 1}]{0}{3}
6   \PstGanttTask[TaskOutsideLabel={Task 2},TaskUnitType=Day]
7     {15}{3} % 3 days starting at day 15
8   \PstGanttTask[TaskStyle=Important,TaskOutsideLabel={Task 3},
9     TaskInsideLabel={\Large\textrm{white}\textbf{Important}}]{2}{5}
10  \PstGanttTask[TaskStyle=NotImportant,
11    TaskOutsideLabel={Task 4}]{4}{2}
12  \PstGanttTask[TaskOutsideLabel={Task 5}]{5}{2}
13 \end{PstGanttChart}

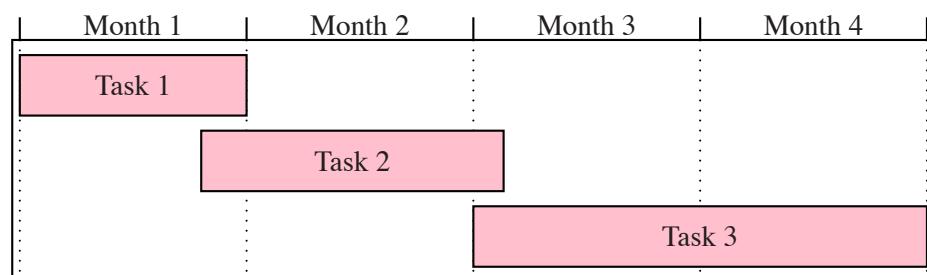
```

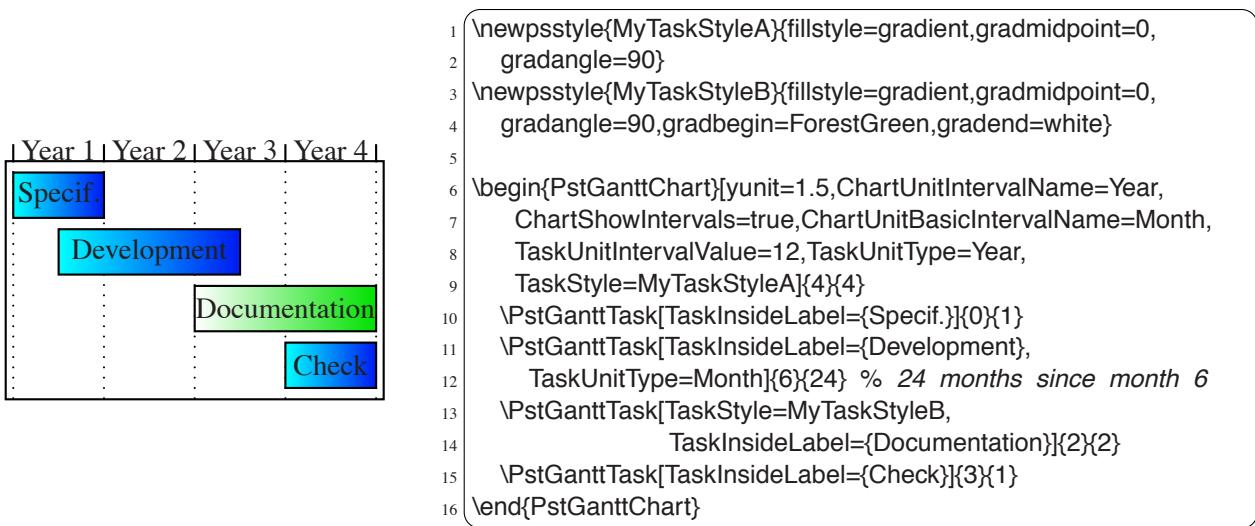


```

1 \newpsstyle{TaskStyle}{fillstyle=solid,fillcolor=Pink}
2
3 \begin{PstGanttChart}[yunit=2,ChartUnitIntervalName=Month,
4   TaskUnitIntervalValue=30,TaskUnitType=Month,
5   ChartShowIntervals,TaskStyle=TaskStyle]{3}{4}
6   \PstGanttTask[TaskInsideLabel={Task 1}]{0}{1}
7   \PstGanttTask[TaskInsideLabel={Task 2},TaskUnitType=Day]
8     {24}{40} % 40 days starting at day 24
9   \PstGanttTask[TaskInsideLabel={Task 3}]{2}{2}
10 \end{PstGanttChart}

```





64.4 Advanced example: overlapped colored surfaces⁵³

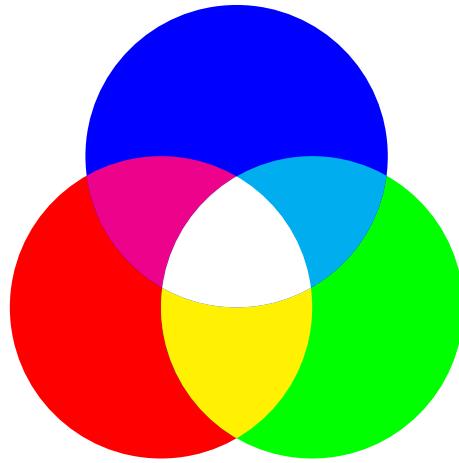


Figure 3: Basic drawing of overlapped circles with colors in additive synthesis

The problem of drawing the picture of the three circles showing the additive synthesis of primary colors (see Figure 3) is very classic and seems simple at the first glance. Nevertheless, there is no high level macro or environment in PStricks which can directly handle this problem.

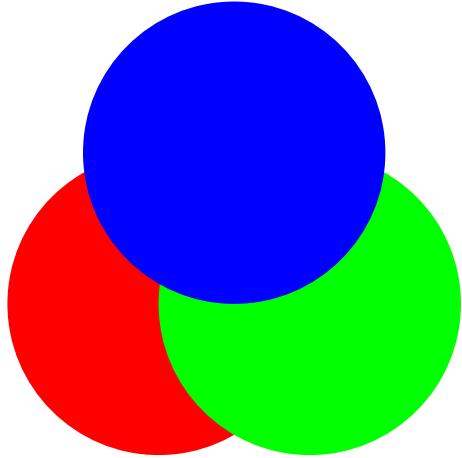
As in many algorithmic problems, a lot of solutions exist, but they are not at all equivalent. Even if here criteria like computing resources (CPU time and memory used) does not matter, the various solutions are very different from the point of view of ease of programming, generality and expressivity.

⁵³This section benefit of some discussions with Manuel Luque in 1999.

We will therefore detail several solutions, from the most basic and *natural* one, but with strong limitations, to more sophisticated and general ones.

In order to simplify the exposition, we will limit our study to the case of three surfaces, as the main difficulty in this problem is in fact due to the necessity to be able to manage several surfaces, which raises complex combinational problems. The code for the general case is much more complex than the one which will be explained here for three surfaces.

We can first draw the three colored circles above each others.



```

1 \psset{linestyle=none,fillstyle=solid}
2 % Drawing of the circles
3 \pscircle[fillcolor=red](0,0){2}
4 \pscircle[fillcolor=green](2,0){2}
5 \pscircle[fillcolor=blue](1,2){2}
```

But, of course, if the surfaces are correct, the colors are not the expected ones, as PostScript cannot do the mixture of them on his own...

We can then draw the filled circles with the primary colors, then using the arc drawing commands **\psarc** and **\psarcn** to redraw the overlapped parts, using the special **\pscustom** command to fill the regions with the additive resulting color.

We obtain the following code (we use polar coordinates here, because this is a lot more easier to do with them):

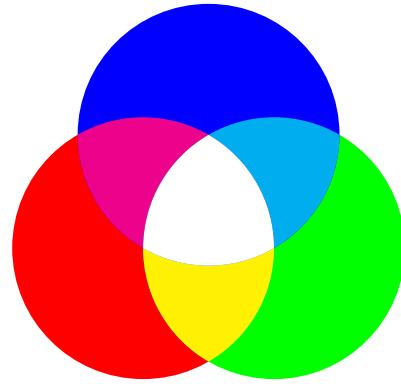
```

1 \SpecialCoor
2 \psset{linestyle=none,fillstyle=solid}
3 %  $\sqrt{1 + 2} = 1.732$ 
4 \pscircle[fillcolor=red](1;210){1.732}%
   % Red circle
5 \pscircle[fillcolor=green](1;330){1.732}%
   % Green circle
6 \pscircle[fillcolor=blue](1;90){1.732}%
   % Blue circle
7 % Yellow intersection of Red and Green circles ( $R + G = Y$ )
8 \pscustom[fillcolor=yellow]{%
9   \psarc(1;-30){1.732}{180}{240} % B
10  \psarc(1;-150){1.732}{-60}{0} % A
11  \psarcn(1;90){1.732}{300}{240}} % C
12 % Cyan intersection of Green and Blue circles ( $G + B = C$ )
13 \pscustom[fillcolor=cyan]{%
14   \psarc(1;90){1.732}{-60}{0} % C
15   \psarc(1;-30){1.732}{60}{120} % B}
```

```

16 \psarcn(1;-150){1.732}{60}{0} % A
17 % Magenta intersection of Red and Blue circles (R + B = M)
18 \pscustom[fillcolor=magenta]{%
19   \psarc(1,90){1.732}{180}{240} % C
20   \psarcn(1;-30){1.732}{180}{120} % B
21   \psarc(1;-150){1.732}{60}{120}} % A
22 % White intersection of Red, Green and Blue circles (R+G+B=W)
23 \pscustom[fillcolor=white]{%
24   \psarc(1,90){1.732}{-120}{-60} % C
25   \psarc(1;-150){1.732}{0}{60} % A
26   \psarc(1;-30){1.732}{120}{180}} % B

```



We can see that this was rather easy to do, but also rather painful, because we must compute all the positions of the points delimiting the various arcs, and we are limited to using *predefined* resulting colors (that is to say that, concerning the colors, we must know the solution before to start to program!) The kind of surfaces is also completely fixed in the program, so we are completely unable to use it to draw the same figure on rhombs or ellipses, for instance, and we are unable to solve the problem on complex closed surfaces such as general polygons or regions enclosed by a general curve.

A very powerful feature of PostScript that we can use with the PSTricks interface (and other general drawing languages) is the *clipping* mechanism. This allows, in a completely general way, drawing a graphic object clipped by other ones. The following code shows how clipping simplifies our little program:

```

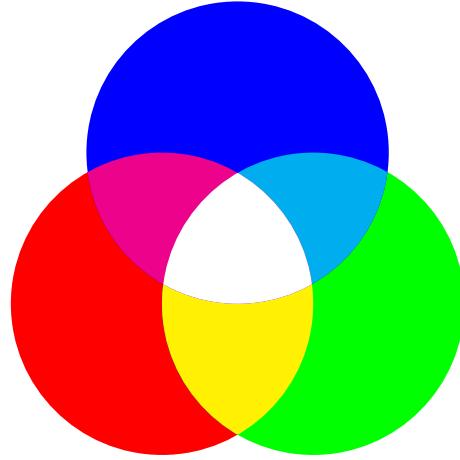
1 \psset{linestyle=none}
2 \def\PstCircleA{\pscircle(0,0){2}}
3 \def\PstCircleB{\pscircle(2,0){2}}
4 \def\PstCircleC{\pscircle(1,2){2}}
5
6 {\psset{fillstyle=solid,fillcolor=red}\PstCircleA} % Red circle
7 {\psset{fillstyle=solid,fillcolor=green}\PstCircleB} % Green circle
8 {\psset{fillstyle=solid,fillcolor=blue}\PstCircleC} % Blue circle
9 %
10 % Yellow intersection of Red and Green circles (R + G = Y)

```

```

11 \psclip{\PstCircleB}
12   \psset{fillstyle=solid,fillcolor=yellow}\PstCircleA
13 \endpsclip
14 % Cyan intersection of Green and Blue circles ( $G + B = C$ )
15 \psclip{\PstCircleB}
16   \psset{fillstyle=solid,fillcolor=cyan}\PstCircleC
17 \endpsclip
18 % Magenta intersection of Red and Blue circles ( $R + B = M$ )
19 \psclip{\PstCircleC}
20   \psset{fillstyle=solid,fillcolor=magenta}\PstCircleA
21 \endpsclip
22 % White intersection of Red, Green and Blue circles ( $R+G+B=W$ )
23 \psclip{\PstCircleB\PstCircleC}
24   \psset{fillstyle=solid,fillcolor=white}\PstCircleA
25 \endpsclip

```



So, the first problem is solved: we have the genericity for the surfaces. This allows management for any closed surfaces, as we no longer have any coordinates to compute. And this solution is far more simple, due to the powerful clipping mechanism. The remaining weakness is that the resulting colors of the overlapped surfaces must still be known externally.

We can build a generalized macro to solve this problem.

```

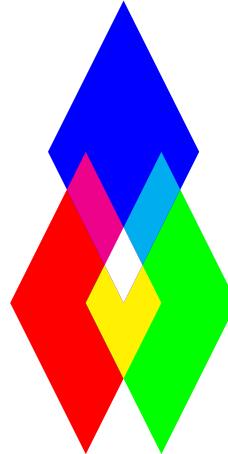
1 % The three surfaces must be defined as \PstSurfaceA,
2 % \PstSurfaceB and \PstSurfaceC
3
4 \def\PstColorSynthesis{%
5   \psset{linestyle=none}%
6   {\psset{fillstyle=solid,fillcolor=red}\PstSurfaceA}%
7   {\psset{fillstyle=solid,fillcolor=green}\PstSurfaceB}%
8   {\psset{fillstyle=solid,fillcolor=blue}\PstSurfaceC}%
9 %
10 % Yellow intersection of Red and Green surfaces ( $R + G = Y$ )
11 \psclip{\PstSurfaceB}
12   \psset{fillstyle=solid,fillcolor=yellow}\PstSurfaceA
13 \endpsclip

```

```

14 % Cyan intersection of Green and Blue surfaces ( $G + B = C$ )
15 \psclip{\PstSurfaceB}
16   \psset{fillstyle=solid,fillcolor=cyan}\PstSurfaceC
17 \endpsclip
18 % Magenta intersection of Red and Blue surfaces ( $R + B = M$ )
19 \psclip{\PstSurfaceC}
20   \psset{fillstyle=solid,fillcolor=magenta}\PstSurfaceA
21 \endpsclip
22 % White intersection of Red, Green and Blue surfaces
23 % ( $R + G + B = W$ )
24 \psclip{\PstSurfaceB\PstSurfaceC}
25   \psset{fillstyle=solid,fillcolor=white}\PstSurfaceA
26 \endpsclip}
27
28 \def\PstSurfaceA{\psdiamond(0,0)(1,2)}
29 \def\PstSurfaceB{\psdiamond(1,0)(1,2)}
30 \def\PstSurfaceC{\psdiamond(0.5,2)(1,2)}
31
32 \PstColorSynthesis

```



Now we can transform it in a PStricks object, as usual.

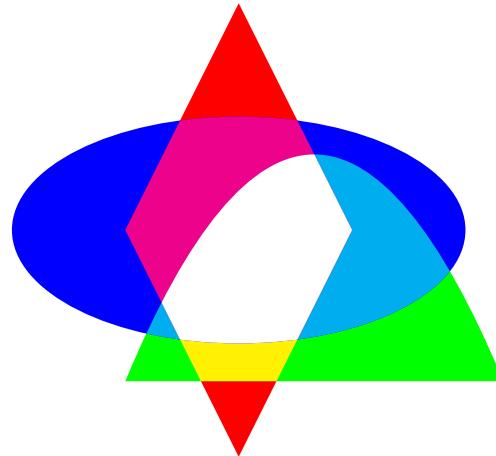
```

1 % Surface parameters
2 \define@key{psset}{SurfaceA}{\def\PstColorSynthesis@SurfaceA{\#1}}
3 \define@key{psset}{SurfaceB}{\def\PstColorSynthesis@SurfaceB{\#1}}
4 \define@key{psset}{SurfaceC}{\def\PstColorSynthesis@SurfaceC{\#1}}
5
6 \def\PstColorSynthesis{\def\pst@par{}\pst@object{PstColorSynthesis}}
7
8 \def\PstColorSynthesis@#1{%
9   \setkeys{psset}{linestyle=none}%
10  \use@par% Assignment of local parameters
11  \bgroup
12    \setkeys{psset}{fillstyle=solid,fillcolor=red}%
13    \PstColorSynthesis@SurfaceA% Red surface
14    \setkeys{psset}{fillcolor=green}%
15    \PstColorSynthesis@SurfaceB% Green surface

```

```

16 \setkeys{psset}{fillcolor=blue}%
17 \PstColorSynthesis@SurfaceC% Blue surface
18 \egroup
19 %
20 % Yellow intersection of Red and Green surfaces (R + G = Y)
21 \psclip{\PstColorSynthesis@SurfaceB}
22 \setkeys{psset}{fillstyle=solid,fillcolor=yellow}%
23 \PstColorSynthesis@SurfaceA
24 \endpsclip
25 % Cyan intersection of Green and Blue surfaces (G + B = C)
26 \psclip{\PstColorSynthesis@SurfaceB}
27 \setkeys{psset}{fillstyle=solid,fillcolor=cyan}%
28 \PstColorSynthesis@SurfaceC
29 \endpsclip
30 % Magenta intersection of Red and Blue surfaces (R + B = M)
31 \psclip{\PstColorSynthesis@SurfaceC}
32 \setkeys{psset}{fillstyle=solid,fillcolor=magenta}%
33 \PstColorSynthesis@SurfaceA
34 \endpsclip
35 % White intersection of Red, Green and Blue surfaces
36 % (R + G + B = W)
37 \psclip{\PstColorSynthesis@SurfaceB\PstColorSynthesis@SurfaceC}
38 \setkeys{psset}{fillstyle=solid,fillcolor=white}%
39 \PstColorSynthesis@SurfaceA
40 \endpsclip}
41 \PstColorSynthesis[SurfaceA=\psdiamond(1.5,3),
42 SurfaceB=\parabola(-1.5,-2)(1,1),SurfaceC=\psellipse(3,1.5)]
```



The next step that we can take is to improve the code of the macro, using some standard \TeX programming techniques. This is not directly related to PSTricks, and we will not get new functionalities, but using a higher level of abstraction will make the program both shorter and easier to modify later.

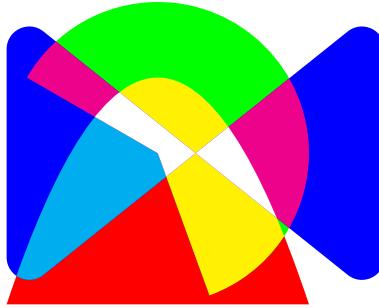
```

1 % The seven color names to use
2 \define@key{psset}{ColorA}{\def\PstColorSynthesis@ColorA{\#1}}
```

```

3 \define@key{psset}{ColorB}{\def\PstColorSynthesis@ColorB{\#1}}
4 \define@key{psset}{ColorC}{\def\PstColorSynthesis@ColorC{\#1}}
5 \define@key{psset}{ColorD}{\def\PstColorSynthesis@ColorD{\#1}}
6 \define@key{psset}{ColorE}{\def\PstColorSynthesis@ColorE{\#1}}
7 \define@key{psset}{ColorF}{\def\PstColorSynthesis@ColorF{\#1}}
8 \define@key{psset}{ColorG}{\def\PstColorSynthesis@ColorG{\#1}}
9
10 \def\PstColorSynthesis@i{%
11 \setkeys{psset}{linestyle=none}%
12 % Assignment of local parameters
13 % The initial surfaces in their original colors
14 \bgroup
15   \setkeys{psset}{fillstyle=solid,fillcolor=\PstColorSynthesis@ColorA}%
16   \PstColorSynthesis@SurfaceA
17   \setkeys{psset}{fillcolor=\PstColorSynthesis@ColorB}%
18   \PstColorSynthesis@SurfaceB
19   \setkeys{psset}{fillcolor=\PstColorSynthesis@ColorC}%
20   \PstColorSynthesis@SurfaceC
21 \egroup
22 % We redraw them in a combinatorial way, with clipping
23 \PstColorSynthesis@ClippedSurfaces{\PstColorSynthesis@SurfaceA}%
24   {\PstColorSynthesis@SurfaceB}{}{\PstColorSynthesis@ColorD}%
25 \PstColorSynthesis@ClippedSurfaces{\PstColorSynthesis@SurfaceA}%
26   {\PstColorSynthesis@SurfaceC}{}{\PstColorSynthesis@ColorE}%
27 \PstColorSynthesis@ClippedSurfaces{\PstColorSynthesis@SurfaceB}%
28   {\PstColorSynthesis@SurfaceC}{}{\PstColorSynthesis@ColorF}%
29 \PstColorSynthesis@ClippedSurfaces{\PstColorSynthesis@SurfaceA}%
30   {\PstColorSynthesis@SurfaceB}{\PstColorSynthesis@SurfaceC}%
31   {\PstColorSynthesis@ColorG}}}
32
33 \def\PstColorSynthesis@ClippedSurfaces#1#2#3#4{%
34 % We draw the first surface #1 in color #4, clipped by the one
35 % or two others #2 and #3
36 \psclip{#2#3}
37   \setkeys{psset}{fillstyle=solid,fillcolor=#4}%
38   #1
39 \endpsclip}
40
41 \PstColorSynthesis[SurfaceA={\parabola(-1,-1)(1,2)},
42 SurfaceB={\pswedge(1,1){2}{-70}{150}},
43 SurfaceC={\pspolygon*[linearc=0.3]{(-1,-1)(-1,3)(4,-1)(4,3)}},
44 ColorA=red,ColorB=green,ColorC=blue,ColorD=yellow,ColorE=cyan,
45 ColorF=magenta,ColorG=white]

```



So, we have already made huge progress! Moreover, the resulting code is far more general, powerful and in fact easier to write as it involves no manual computations of intersection regions. It is no longer than our first working version. Nevertheless, a major weakness remains: the mixed colors must be manually computed externally and provided as parameters. To solve this problem, we must use other \TeX programming techniques.⁵⁴

```

1 % Additive color synthesis (correct results only for sums
2 % of RGB components <= 1)
3 % Colors must be defined according the RGB model
4 % (so, for instance, to use "black", it must be redefined in RGB)
5
6 \def\PstColorSynthesis@#1{%
7   \setkeys{psset}{linestyle=none}%
8   \use@par% Assignment of local parameters
9   \bgroup
10    \setkeys{psset}{fillstyle=solid,fillcolor=\PstColorSynthesis@ColorA}%
11    \PstColorSynthesis@SurfaceA
12    \setkeys{psset}{fillcolor=\PstColorSynthesis@ColorB}%
13    \PstColorSynthesis@SurfaceB
14    \setkeys{psset}{fillcolor=\PstColorSynthesis@ColorC}%
15    \PstColorSynthesis@SurfaceC
16  \egroup
17  % We redraw them in a combinatorial way, with clipping,
18  % and computing the resulting color of overlapped surfaces
19  % two by two then the three together
20  \PstColorSynthesis@ClippedSurfaces{A}{B}{NoSurface}%
21  \PstColorSynthesis@ClippedSurfaces{A}{C}{NoSurface}%
22  \PstColorSynthesis@ClippedSurfaces{B}{C}{NoSurface}%
23  \PstColorSynthesis@ClippedSurfaces{A}{B}{C}}}

24
25 % For the cases where there are only two surfaces
26 % in the combination
27 \def\PstColorSynthesis@ColorNoSurface{}

28
29 % We draw the first surface #1 clipped by the one
30 % or two others #2 and #3, computing its resulting color

```

⁵⁴Note that the proposed solution here is based on the `color` package, which can be used with PLAIN \TeX and \LaTeX , but with Con \TeX t, modifications must be made to implement color management.

```

31 \def\PstColorSynthesis@ClippedSurfaces#1#2#3{%
32 % We compute the "mixed" color, component by component
33 \def\PstColorSynthesis@MixedColorR{0}%
34 \def\PstColorSynthesis@MixedColorG{0}%
35 \def\PstColorSynthesis@MixedColorB{0}%
36 \PstColorSynthesis@MixedColor{%
37   \csname PstColorSynthesis@Color#1\endcsname}%
38 \PstColorSynthesis@MixedColor{%
39   \csname PstColorSynthesis@Color#2\endcsname}%
40 \PstColorSynthesis@MixedColor{%
41   \csname PstColorSynthesis@Color#3\endcsname}%
42 % We draw the first surface, clipped by the other ones
43 \psclip{\csname PstColorSynthesis@Surface#2\endcsname}%
44   \csname PstColorSynthesis@Surface#3\endcsname}%
45   \definecolor{MixedColor}{rgb}{\PstColorSynthesis@MixedColorR,
46     \PstColorSynthesis@MixedColorG,
47     \PstColorSynthesis@MixedColorB}%
48   \setkeys{psset}{fillstyle=solid,fillcolor=MixedColor}%
49   \csname PstColorSynthesis@Surface#1\endcsname
50 \endpsclip}

51 \def\PstColorSynthesis@MixedColor#1{%
52 \edef@tempa{#1}%
53 \ifx@tempa@empty
54 \else
55   % We ask for the values of the 3 components of the color.
56   % This is the difficult line:
57   % - in the "color" package, a color is defined by the macro
58   %   \color@ColorName,
59   % - so, we must build the macro name \color@#1 and call
60   %   the macro \PstColorSynthesis@MixedColor@i with the
61   %   result of the execution of it.
62   %
63   % Note also that, if we want to manage too the "cmyk" color
64   % model, this macro and the next one must be modified, as
65   % \color@ColorName will return one argument more:
66   % \pst@expandafter\PstColorSynthesis@MixedColor@i{%
67   %   \csname\string\color @#1\endcsname} \@nil
68   %
69   %
70   % \def\PstColorSynthesis@MixedColor@i#1 #2 #3 #4 #5@nil{%
71   %
72   \pst@expandafter\PstColorSynthesis@MixedColor@i{%
73     \csname\string\color@#1\endcsname}\@nil
74 \fi}

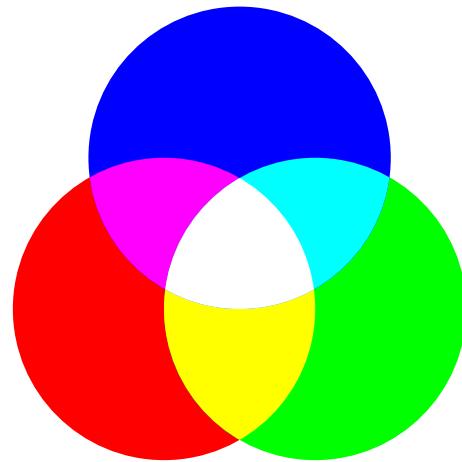
75 \def\PstColorSynthesis@MixedColor@i#1 #2 #3 #4 #5@nil{%
76 % We receive the name of the color model ("rgb" is expected
77 % here) and the three color components.
78 % We add the values to the respective components of the new
79 % color to compute.
80 \PstColorSynthesis@MixedColor@ii{%
81   \PstColorSynthesis@MixedColorR{#2}%
82   \PstColorSynthesis@MixedColor@ii{%

```

```

84 \PstColorSynthesis@MixedColorG}{#3}%
85 \PstColorSynthesis@MixedColor@ii{%
86   \PstColorSynthesis@MixedColorB}{#4}%
87
88 \def\PstColorSynthesis@MixedColor@ii#1#2{%
89 % As these values are real numbers, we use a dimension
90 % register, then we assign the computed value in a macro,
91 % converting it from a dimension to a number
92 \pst@dimg=#1\p@
93 \advance\pst@dimg by #2\p@
94 \pst@dimtonum{\pst@dimg}{#1}%
95
96 \PstColorSynthesis[SurfaceA={\pscircle{2}},
97 SurfaceB={\pscircle(2,0){2}},SurfaceC={\pscircle(1,2){2}},
98 ColorA=red,ColorB=green,ColorC=blue]

```



What is still missing is correct automatic computation of the mixed colors, according to the rules of the *additive color synthesis*, in cases where the sum of individual color components is greater than 1.⁵⁵

```

1 % We draw the first surface #1 clipped by the one
2 % or two others #2 and #3, computing it resulting color
3 \def\PstColorSynthesis@ClippedSurfaces#1#2#3{%
4 % We compute the "mixed" color, component by component

```

⁵⁵For a generalized solution to the drawing of overlapped surfaces, you can see the code of the ‘pst-csyn’ contribution package [56], from Denis Girou and Manuel Luque which implement:

- several synthesis managements (additive, subtractive, average),
- several color models (RGB and CMYK),
- the management of any number of surfaces (which is in fact the main difficulty in this problem, as it open a combinational problem.),
- support both the colors management of the `color` package with PLAIN \TeX and \LaTeX and the one of \ConTeXt .

```

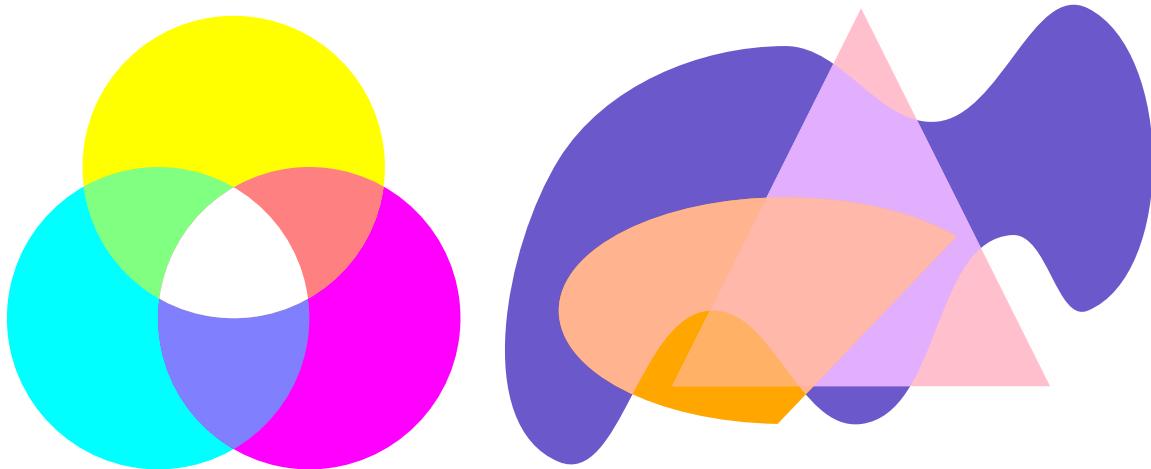
5  \def\PstColorSynthesis@MixedColorR{0}%
6  \def\PstColorSynthesis@MixedColorG{0}%
7  \def\PstColorSynthesis@MixedColorB{0}%
8  % \pst@dimd will contain the maximum value of the three
9  % components.
10 \pst@dimd=\z@
11 \PstColorSynthesis@MixedColor{%
12   \csname PstColorSynthesis@Color#1\endcsname}%
13 \PstColorSynthesis@MixedColor{%
14   \csname PstColorSynthesis@Color#2\endcsname}%
15 \PstColorSynthesis@MixedColor{%
16   \csname PstColorSynthesis@Color#3\endcsname}%
17 % We must test if the maximum of the new computed
18 % component is greater than 1, in which case we divide all
19 % the three components by this value for additive synthesis.
20 \ifdim\pst@dimd>\@ne\p@
21   \PstColorSynthesis@MixedColor@iii{%
22     \PstColorSynthesis@MixedColorR}%
23   \PstColorSynthesis@MixedColor@iii{%
24     \PstColorSynthesis@MixedColorG}%
25   \PstColorSynthesis@MixedColor@iii{%
26     \PstColorSynthesis@MixedColorB}%
27 \fi
28 % We draw the first surface, clipped by the other ones
29 \psclip{\csname PstColorSynthesis@Surface#2\endcsname}%
30   \csname PstColorSynthesis@Surface#3\endcsname}%
31   \definecolor{MixedColor}{rgb}{\PstColorSynthesis@MixedColorR,
32     \PstColorSynthesis@MixedColorG,
33     \PstColorSynthesis@MixedColorB}%
34   \setkeys{psset}{fillstyle=solid,fillcolor=MixedColor}%
35   \csname PstColorSynthesis@Surface#1\endcsname
36 \endpsclip}
37
38 \def\PstColorSynthesis@MixedColor@i#1 #2 #3 #4@nil{%
39 % We receive the name of the color model ("rgb" is expected
40 % here and the three color components.)
41 % We add the values to the respective components of the new
42 % color to compute.
43 \PstColorSynthesis@MixedColor@ii{%
44   \PstColorSynthesis@MixedColorR}{#2}%
45 \PstColorSynthesis@MixedColor@ii{%
46   \PstColorSynthesis@MixedColorG}{#3}%
47 \PstColorSynthesis@MixedColor@ii{%
48   \PstColorSynthesis@MixedColorB}{#4}}
49
50 \def\PstColorSynthesis@MixedColor@ii#1#2{%
51 % As these values are real numbers, we use dimension
52 % registers, then we assign the computed value in a macro,
53 % converting it from a dimension to a number.
54 % We also keep in \pst@dimd the maximum of the values.
55 \pst@dimg=#1\p@
56 \advance\pst@dimg by #2\p@
57 \pst@dimtonum{\pst@dimg}{#1}%

```

```

58 \ifdim\pst@dimg>\pst@dimd
59   \pst@dimd=\pst@dimg
60 \fi}
61
62 \def\PstColorSynthesis@MixedColor@iii#1{%
63   \pst@divide{\#1\p@}{\pst@dimd}\#1}%
64 % We must take care of possible rounding problems with
65 % \pst@divide (for instance, 1.8/1.8 give 1.0001)
66 \ifdim#1\p@>@\ne\p@
67   \def#1{1}%
68 \fi}
69
70 % We redefine Cyan, Magenta and Yellow in the "rgb" model
71 \definecolor{Cyan}{rgb}{0,1,1}%
72 \definecolor{Magenta}{rgb}{1,0,1}%
73 \definecolor{Yellow}{rgb}{1,1,0}%
74 \PstColorSynthesis[SurfaceA=\pscircle{2},
75   SurfaceB=\pscircle(2,0){2},SurfaceC=\pscircle(1,2){2},
76   ColorA=Cyan,ColorB=Magenta,ColorC=Yellow]
77 \PstColorSynthesis[SurfaceA=\psccurve(-3,1)(0,2.5)(2,1.5)(4,3)(4,-1)
78   (3,0)(1,-2.5)(-1,-1)(-3,-3)},
79   SurfaceB=\psellipticarc(0,-1)(3,1.5){41}{-92},
80   SurfaceC=\psttriangle(1,-2)(5,5),
81   ColorA=SlateBlue,ColorB=Orange,ColorC=Pink]

```



64.5 Advanced example: three dimensional lighten effect

In this example, we will mainly emphasize how to use doc format, which is a very convenient way to organize and distribute a package, and how to define and use a new PostScript header file, relative to a specialized package. The main purpose to do that is to load some PostScript code only one time and to reuse it as many times as needed in some \TeX macros, without duplicate it. This is the general way of how PSTricks works. The new header

file must be accessible to the dvi converter, or directly to the compiler if it contain itself a PostScript interpreter, like someones have.

We will illustrate this way to proceed on an example which has the only behavior to produce a visual effect, similar to a light on a three dimensional frame of objects. This could be applied to both characters and PSTricks curves. The idea come from a short PostScript code from Peter Kleiweg [36], which create a lighten effect on characters. We interface this PostScript code with \TeX ,⁵⁶ give it a PSTricks interface, and extend it to opened or closed curves –but this does not work with most of the line parameters (arrows, linestyle, doubleline, etc.)

The code is short but not very simple. Nevertheless, most readers can forgot the difficult parts, just looking at the architecture of the code and at the interaction between the \TeX and the PostScript macros. The \TeX macros are for a large part an adaptation of the **\pscharpath** macro from the ‘pst-char’ package (see Section 59.)

In addition, we use here the **doc** format [11] (by convention, these files use the **.dtx** suffix), which allow to embed in only one file all the various source files of a package, with the user’s documentation and the internal documentation of the source code. This is a \LaTeX package, but \LaTeX is only required to generate the documentation and the package itself can be a generic one, working both with PLAIN \TeX and ConTeXt, as this is the case here.

First we give the driver file ‘**pst-li3d.ins**’, which will generate the various files of the package when compiling it by \TeX . Three files will be created here, **pst-li3d.sty** (the \LaTeX wrapper), **pst-li3d.tex** (the generic \TeX file) and **pst-li3d.pro** (the PostScript header file.) Here, we must take care to do not generate an **\endinput** line at the end of the PostScript file, which is the default behavior, relevant for all \TeX files, but this would cause an error for a non \TeX file as this one (we use the **\usepostamble** macro to change this behavior.)

```

1 %% 'pst-li3d.ins'
2 %%
3 %% Docstrip installation instruction file for docstyle 'pst-li3d'
4 %%
5 %% Denis Girou (CNRS/IDRIS - France) <Denis.Girou@idris.fr>
6 %% and Peter Kleiweg (Rijksuniversiteit Groningen - Nederlands)
7 %% <kleiweg@let.rug.nl>
8 %%
9 %% July 10, 2003
10 \def\batchfile{pst-li3d.ins}

```

⁵⁶We keep the same algorithm, but we change in some place the order of the computations to avoid to generate overflow computations in PostScript, which frequently arrived otherwise with the numerical values manipulated by \TeX on PostScript fonts. Take care that nevertheless such overflows could still arrive

```

12 \input docstrip.tex
13 \keepsilent
14 \Msg{*** Generating the 'pst-li3d' package ***}
15 \askforoverwritefalse
16 \generate{\file{pst-li3d.tex}{\from{pst-li3d.dtx}{pst-li3d}}}
17 \generate{\file{pst-li3d.sty}{\from{pst-li3d.dtx}{\latex-wrapper}}}
18
19 % We must not write the "\endinput" line at the end
20 % of this external file!
21 \def\EpsFilePostamble{\MetaPrefix\space End of file `outFileName'.}
22 \usepostamble\EpsFilePostamble
23 \generate{\file{pst-li3d.pro}{\from{pst-li3d.dtx}{postscript-header}}}
24
25 \ifToplevel{%
26 \Msg{*****}
27 \Msg{*}
28 \Msg{* To finish the installation you have to move the files}
29 \Msg{* pst-li3d.sty and pst-li3d.tex in a directory/folder searched}
30 \Msg{* by TeX and the file pst-li3d.pro in a directory/folder}
31 \Msg{* searched by the DVI converter or the compiler, depending}
32 \Msg{* of your installation.}
33 \Msg{*}
34 \Msg{* To produce the documentation, run the file 'pst-li3d.dtx'}
35 \Msg{* through LaTeX.}
36 \Msg{*}
37 \Msg{* If you require the commented code, desactivating the}
38 \Msg{* OnlyDescription macro, you must recompile, execute:}
39 \Msg{* 'makeindex -s gind.ist pst-li3d'}
40 \Msg{* 'makeindex -s gglo.ist -o pst-li3d.gls pst-li3d.glo'}
41 \Msg{* and recompile.}
42 \Msg{*}
43 \Msg{*****}
44 }
45
46 \endinput
47 %%
48 %% End of file 'pst-li3d.ins'

```

Then the core file, called ‘pst-li3d.dtx’. Note that the comment prefix for this file (not for the ones that will be generated!) is $\wedge\wedge A$.

```

1 % \iffalse meta-comment, etc.
2 %%
3 %% Package 'pst-li3d.dtx'
4 %%
5 %% Denis Girou (CNRS/IDRIS - France) <Denis.Girou@idris.fr>
6 %% and Peter Kleiweg (Rijksuniversiteit Groningen - Nederlands)
7 %% <kleiweg@let.rug.nl>
8 %%
9 %% July 10, 2003
10 %%
11 %% This program can be redistributed and/or modified under

```

```

12 %% the terms of the LaTeX Project Public License Distributed
13 %% from CTAN archives in directory macros/latex/base/ppl.txt.
14 %%
15 %% DESCRIPTION:
16 %%   'pst-li3d' is a PSTricks package for three dimensional
17 %%   lighten effect on characters and PSTricks graphics.
18 %%
19 % \fi

```

The `\changes` macro allow to describe the various releases of the package.

```

20 % \changes{v1.0}{2003/07/10}{First public release.}

```

The `\CheckSum` macro offer a simple way to verify the integrity of the .dtx file. It is based on the number of macros used in the file. The exact value will be given after a compilation of the file by L^AT_EX, and is to put as the argument of the macro.

```

21 % \CheckSum{121}

```

The `\DoNotIndex` macros allow to give the list of the macros used in the file which are not to be used as entries to generate the index (for the documentation of the source code), as this index is supposed to refer mainly to the new macros defined in the package. And settings the `IndexColumns` counter to 2 require to format the index on two columns.

```

22 % \DoNotIndex{@}
23 % \DoNotIndex{\begin,\begin@ClosedObj,\box}
24 % ...
25 % \DoNotIndex{\use@par,\use@pscode,\usepackage}
26 %
27 % ^^A For the index and changes log (if source code printed)
28 % \setcounter{IndexColumns}{2}
29 % \setlength{\columnseprule}{0.6pt}

```

We then define some utility macros convenient to format the documentation of the package. This is mainly some *example* macros,⁵⁷ which allow to include only one time the source code of some examples and to format it twice, one time in verbatim mode showing the code and another time as the result produced by the compilation of this code. Such macros are essential to guarantee that the two ones will be synchronised after various updates of the documentation.

⁵⁷The ones we use here are built above the ‘fancyvrb’ package [16].

```

% ^A Utility macros
%
% \newcommand{\PstLightThreeDPackage}{\textsf{pst-li3d}}
%
% ^A A list of options for a package/class (from ltugboat.cls)
% \newenvironment{optlist}{\begin{description}}%
%   \renewcommand\makelabel[1]{%
%     \descriptionlabel{\mdseries\textsf{\#1}}\%
%     \itemsep0.25\itemsep}\%
%   \end{description}}%
%
% \makeatletter
%
% ^A Example macros
%

```

Then we insert all the user's documentation of the package, not included here, as we will add it later, but formatted.

```

% ^A Beginning of the documentation itself
%
% \title{The \PstLightThreeDPackage{} package\\
%         A PSTricks package for three dimensional\\
%         lighten effect on characters and PSTricks graphics}
% \author{Denis \textsc{Girou}\\
%         ...\\
%         and Peter \textsc{Kleiweg}\\
%         ...}
% \date{Version 1.0\July 10, 2003\\
%       {\small Documentation revised July 10, 2003}}
%
% \maketitle
%
% \begin{abstract}
%   This package allow to add a three dimensional lighten
%   effect on characters (PostScript fonts), using the
%   \cs{PstLightThreeDText} macro, and curves (opened or closed),
%   using the \cs{PstLightThreeDGraphic} macro, with various
%   customization parameters.
% \end{abstract}
%
% \tableofcontents
%
% \section{Introduction}
% ...
% \section{Usage}
%
% \section{Examples}
% ...
% \StopEventually{}
```

```

76 %  

77 % ^A ..... End of the documentation part .....
```

4 Driver file

Then we include the driver file which will be used to generate the documentation.

```

1 % \section{Driver file}  

2 %  

3 % The next bit of code contains the documentation driver file  

4 % for \TeX{}, i.e., the file that will produce the documentation  

5 % you are currently reading. It will be extracted from this file  

6 % by the \texttt{docstrip} program.  

7 %  

8 % \begin{macrocode}  

9 %<*driver>  

10 \documentclass{ltxdoc}  

11 \GetFileInfo{pst-li3d.dtx}  

12 \usepackage{fancyvrb}  

13 \usepackage{multido}  

14 \usepackage{pstcol}  

15 \usepackage{pst-grad}  

16 \usepackage{pst-li3d}  

17 \usepackage{pst-plot}  

18 \usepackage{pst-tree}  

19 \usepackage{url}  

20 \definecolor{DarkGreen}{cmyk}{1,0,1,0.8}  

21 \definecolor{Gold} {rgb}{1,0.84,0}  

22 \definecolor{Violet} {cmyk}{0.79,0.88,0,0}  

23 \EnableCrossrefs  

24 \CodelineIndex  

25 \RecordChanges  

26 \OnlyDescription % Comment it for implementation details  

27 \hbadness=7000 % Over and under full box warnings  

28 \hfuzz=3pt  

29 \begin{document}  

30   \DocInput{pst-li3d.dtx}  

31 \end{document}  

32 %</driver>  

33 % \end{macrocode}
```

5 ‘*pst-li3d*’ *LATEX* wrapper

Then we define the short *LATEX* wrapper file, which will be useful for *LATEX* users.

```

1 % \section{\PstLightThreeDPackage{} \LaTeX{} wrapper}  

2 %  

3 % \begin{macrocode}
```

```

4 %<*latex-wrapper>
5 \ProvidesPackage{pst-li3d}[2003/07/10 package wrapper
6                               for PSTRicks pst-li3d.tex]
7 \input{pst-li3d}
8 %</latex-wrapper>
9 %     \end{macrocode}

```

And now the generic T_EX (and PSTRicks) package. The source comments are inside the file, but we just repeat them here to have also them formatted, as when the documented source code is generated.

6 ‘*pst-li3d*’ code

```

1 % \section{\PstLightThreeDPackage{} code}
2 %
3 %     \begin{macrocode}
4 %<*pst-li3d>
5 %     \end{macrocode}

```

6.1 *Preamble*

Who we are.

```

1 % \subsection{Preamble}
2 %
3 % Who we are.
4 %
5 % \begin{macrocode}
6 \def\FileVersion{1.0}
7 \def\FileDate{2003/07/10}
8 \message{Pst-Light3d' \FileVersion, \FileDate\space
9          (Denis Girou and Peter Kleiweg)}
10 \csname PSTLightThreeDLoaded\endcsname
11 \let\PSTLightThreeDLoaded\endinput
12 % \end{macrocode}

```

Require the PSTRicks package.

```

1 % Require the PSTRicks package.
2 %
3 % \begin{macrocode}
4 \ifx\PSTRicksLoaded\endinput\else\input{pstricks}\fi
5 % \end{macrocode}

```

David CARLISLE interface to the ‘keyval’ package.

```

1 % David \textsc{Carlisle} interface to the ‘\textsf{keyval}’
2 % package.
3 %

```

```

4 %     \begin{macrocode}
5 \input{pst-key}
6 %     \end{macrocode}

```

Catcodes changes.

```

1 %   Catcodes changes.
2 %
3 %   \begin{macrocode}
4 \edef\PstAtCode{\the\catcode`@}
5 \catcode`@=11\relax
6 %   \end{macrocode}

```

We load the PostScript header file.

```

1 %   We load the PostScript header file.
2 %
3 %   \begin{macrocode}
4 \pstheader{pst-li3d.pro}
5 %   \end{macrocode}

```

6.2 Definition of the parameters

`LightThreeDXLength` will be the horizontal length of the frame added. It is a *length* value.

```

1 % \subsection{Definition of the parameters}
2 %
3 % \texttt{LightThreeDXLength} will be the horizontal length of the
4 % frame added. It is a \emph{length} value.
5 %
6 %   \begin{macrocode}
7 \define@key{psset}{LightThreeDXLength}{%
8 \pst@getlength{#1}\PstLightThreeD@XLength}%
9 %   \end{macrocode}

```

`LightThreeDYLength` will be the vertical length of the frame added. It is a *length* value.

```

1 % \texttt{LightThreeDYLength} will be the vertical length of the
2 % frame added. It is a \emph{length} value.
3 %
4 %   \begin{macrocode}
5 \define@key{psset}{LightThreeDYLength}{%
6 \pst@getlength{#1}\PstLightThreeD@YLength}%
7 %   \end{macrocode}

```

`LightThreeDLength` will define both the horizontal and the vertical length of the frame added. It is a *length* value.

```

1 % \texttt{LightThreeDLength} will define both the horizontal and
2 % the vertical length of the frame added. It is a \emph{length}
3 % value.
4 %
5 % \begin{macrocode}
6 \define@key{psset}{LightThreeDLength}{%
7 \pst@getlength{#1}\PstLightThreeD@XLength}%
8 \pst@getlength{#1}\PstLightThreeD@YLength}%
9 % \end{macrocode}

```

LightThreeSteps will be the number of steps for the light effect. It works like a resolution parameter. It is an *integer* value.

```

1 % \texttt{LightThreeSteps} will be the number of steps for the
2 % light effect. It works like a resolution parameter. It is an
3 % \emph{integer} value.
4 %
5 % \begin{macrocode}
6 \define@key{psset}{LightThreeDSteps}{%
7 \pst@getint{#1}\PstLightThreeD@Steps}}%
8 % \end{macrocode}

```

LightThreeDAngle will be the angle used. It will act as an angle for the light effect. It is a *real* value used as an *angle*.

```

1 % \texttt{LightThreeDAngle} will be the angle used. It will act as
2 % an angle for the light effect. It is a \emph{real} value used as
3 % an \emph{angle}.
4 %
5 % \begin{macrocode}
6 \define@key{psset}{LightThreeDAngle}{%
7 \pst@getangle{#1}\PstLightThreeD@LightAngle}}%
8 % \end{macrocode}

```

LightThreeDColorPsCommand will be the PostScript sequence of commands which will define the color of the frame added, which must use the last value of the PostScript stack. Good results are obtained when this value is used with the `setgray` operator or as the Saturation or Brightness component of the `sethsbcolor` one, using the HSB color model. It is a *string* value.

```

1 % \texttt{LightThreeDColorPsCommand} will be the PostScript
2 % sequence of commands which will define the color of the frame
3 % added, which must use the last value of the PostScript stack.
4 % Good results are obtained when this value is used with the
5 % \texttt{setgray} operator or as the Saturation or Brightness
6 % component of the \texttt{sethsbcolor} one, using the HSB color
7 % model.
8 % It is a \emph{string} value.

```

```

9  %
10 %     \begin{macrocode}
11 \define@key{psset}{LightThreeDColorPsCommand}{%
12 \edef\PstLightThreeD@ColorPsCommand{\#1}}
13 %     \end{macrocode}

```

Next, we set the default values for all these new parameters.

```

1 %   Next, we set the default values for all these new parameters.
2 %
3 %     \begin{macrocode}
4 \setkeys{psset}{%
5 LightThreeDXLength=0.2,LightThreeDYLength=0.3,
6 LightThreeDSteps=40,LightThreeDAngle=45,
7 LightThreeDColorPsCommand=2.5 div setgray}
8 %     \end{macrocode}

```

6.3 Main macros

The general \PstLightThreeDGraphic macro to apply the lighten effect on a graphic (an opened or closed curve). This is a PSTRicks object.

```

1 % \subsection{Main macros}
2 %
3 %   The general \cs{PstLightThreeDGraphic} macro to apply the
4 %   lighten effect on a graphic (an opened or closed curve).
5 %   This is a PSTRicks object.
6 %
7 % \begin{macro}{\PstLightThreeDGraphic}
8 %   \begin{macrocode}
9 \def\PstLightThreeDGraphic{%
10 \def\pst@par{}%
11 \pst@object{PstLightThreeDGraphic}%
12 %   \end{macrocode}
13 % \end{macro}

```

Then we define it auxiliary macro which will handle the parameters, if some are used, initialize the PostScript environment and redefine the stroke PostScript operator calling two times the LightThreeDPathForAll PostScript macro defined in the header file.

```

1 %   Then we define it auxiliary macro which will handle the
2 %   parameters, if some are used, initialize the PostScript
3 %   environment and redefine the \texttt{stroke} PostScript
4 %   operator calling two times the \texttt{LightThreeDPathForAll}
5 %   PostScript macro defined in the header file.
6 %
7 % \begin{macro}{\PstLightThreeDGraphic@i}
8 %   \begin{macrocode}
9 \def\PstLightThreeDGraphic@i{%

```

```

10 \pst@makebox{%
11   \PstLightThreeD@i{%
12     /LightThreeDDXa LightThreeDDX def
13     /LightThreeDDYa LightThreeDDY def
14     /stroke {gsave reversepath LightThreeDPathForAll grestore
15       LightThreeDPathForAll } def}{end}}}
16   \end{macrocode}
17 \end{macro}

```

Now, the general `\PstLightThreeDText` macro to apply the lighten effect on a text (using PostScript fonts). This is a PSTricks object too.

```

1 % Now, the general \cs{PstLightThreeDText} macro to apply the
2 % lighten effect on a text (using PostScript fonts). This is a
3 % PSTricks object too.
4 %
5 % \begin{macro}{\PstLightThreeDText}
6 %   \begin{macrocode}
7 \def\PstLightThreeDText{\def\pst@par{} \pst@object{PstLightThreeDText}}
8 %   \end{macrocode}
9 % \end{macro}

```

Then we define it auxiliary macro which will handle the parameters, if some are used, initialize the PostScript environment, keep the content of the `show` PostScript operator, redefine it, calling the `\texttt{LightThreeDPathForAll}` PostScript macro defined in the header file, redraw a second time the outline path using the `\texttt{charpath}` operator, and restore the old content of the `show` operator. In fact, this code is very similar to the one of the `\pscharpath` macro from the ‘`pst-char`’ package.

```

1 % Then we define it auxiliary macro which will handle the
2 % parameters, if some are used, initialize the PostScript
3 % environment, keep the content of the \texttt{show} PostScript
4 % operator, redefine it, calling the \texttt{LightThreeDPathForAll}
5 % PostScript macro defined in the header file, redraw a second
6 % time the outline path using the \texttt{charpath} operator,
7 % and restore the old content of the \texttt{show} operator. In fact,
8 % this code is very similar to the one of the \cs{pscharpath}
9 % macro from the ‘\textsf{pst-char}’ package.
10 %
11 % \begin{macro}{\PstLightThreeDText@i}
12 %   \begin{macrocode}
13 \def\PstLightThreeDText@i{%
14   \pst@makebox{%
15     \PstLightThreeD@i{%
16       /LightThreeDDXa LightThreeDDX Resolution 100 div mul def
17       /LightThreeDDYa LightThreeDDY VResolution 100 div mul def
18       /tx@LightThreeDSavedShow /show load def
19       /show {
20         dup

```

```

21      gsave
22      false charpath
23      reversepath
24      LightThreeDPathForAll
25      grestore
26      true charpath} def}
27      {/show /tx@LightThreeDSavedShow load def
28      end}%
29      \begin@ClosedObj
30      \def\use@pscode{%
31      \pst@Verb{%
32      gsave
33      \tx@STV
34      \pst@code
35      grestore
36      CP newpath moveto}%
37      \gdef\pst@code{}%
38      \end@ClosedObj}}%
39      % \end{macrocode}
40      % \end{macro}

```

6.4 Auxiliary macro

Both the \PstLightThreeDGraphic and \PstLightThreeDText macros call the auxiliary \PstLightThreeD@i macro, which assign the local PStricks parameters initialize the various PostScript variables which depend of the PStricks parameters that the user can change, draw the content (text or PStricks graphic) and restore some PostScript operator, if needed.

```

1  % \subsection{Auxiliary macro}
2  %
3  % Both the \cs{PstLightThreeDGraphic} and
4  % \cs{PstLightThreeDText} macros call the auxiliary
5  % \cs{PstLightThreeD@i} macro, which assign the local PStricks
6  % parameters initialize the various PostScript variables which
7  % depend of the PStricks parameters that the user can change,
8  % draw the content (text or PStricks graphic) and restore some
9  % PostScript operator, if needed.
10 %
11 % \begin{macro}{\PstLightThreeD@i}
12 %   \begin{macrocode}
13 \def\PstLightThreeD@i#1#2{%
14   % \end{macrocode}

```

Assignment of local parameters.

```

1  % Assignment of local parameters.
2  %
3  % \begin{macrocode}
4  \use@par
5  % \end{macrocode}

```

Initialization of the various PostScript variables which depend of the PStricks parameters that the user can change.

```

1 % Initialization of the various PostScript variables which
2 % depend of the PStricks parameters that the user can change.
3 %
4 % \begin{macrocode}
5 \leavevmode
6 \pstVerb{%
7   tx@LightThreeDDict begin
8     /LightThreeDDX \PstLightThreeD@XLength\space def
9     /LightThreeDDY \PstLightThreeD@YLength\space neg def
10    /LightThreeDSteps \PstLightThreeD@Steps def
11    /LightThreeDAngle \PstLightThreeD@LightAngle def
12    /LightThreeDColorPsCommand
13      { \PstLightThreeD@ColorPsCommand} def
14    /LightThreeDMINangle
15      LightThreeDDY LightThreeDDX atan 180 sub def
16    /LightThreeDMAngle
17      LightThreeDDY LightThreeDDX atan def
18  #1}%
19 % \end{macrocode}

```

Draw the content (text or PStricks graphic).

```

1 % Draw the content (text or PStricks graphic).
2 %
3 % \begin{macrocode}
4 \hbox{\box\pst@hbox}%
5 % \end{macrocode}

```

Restoration, if needed, of some PostScript operator (for texts, the show one, in fact).

```

1 % Restoration, if needed, of some PostScript operator
2 % (for texts, the "show" one, in fact).
3 %
4 % \begin{macrocode}
5 \pstVerb{#2}}
6 % \end{macrocode}
7 % \end{macro}

```

6.5 Closing

Catcodes restoration.

```

1 % \subsection{Closing}
2 %
3 % Catcodes restoration.
4 %

```

```

5 %     \begin{macrocode}
6 \catcode`@=\PstAtCode\relax
7 %     \end{macrocode}

```

```

1 %     \begin{macrocode}
2 %</pst-li3d>
3 %     \end{macrocode}

```

And now, to finish, the PostScript header file, with the PostScript macros used,⁵⁸ which will be put in an header file ‘pst-3dli.pro’.⁵⁹ We define our new PostScript macros inside a personal dictionary. Four macros are defined, to be able to redefine later the standard basic moveto, lineto, curveto and closepath operators with the pathforall macro, which will allow to draw the current paths in the new way that we program here.

7 ‘*pst-li3d*’ PostScript header file

First, identification and references.

```

1 %     \begin{macrocode}
2 %<*postscript-header>
3 %     \end{macrocode}

```

```

1 %     \begin{macrocode}
2 %!
3 % PostScript header file pst-li3d.pro
4 % Version 1.0, 2003/07/10
5 %
6 % Adapted from Peter Kleiweg <kleiweg@let.rug.nl>
7 % See http://odur.let.rug.nl/~kleiweg/postscript/postscript.html
8 %     (file depth.ps)
9 %
10 %    \end{macrocode}

```

The dictionary for this package.

```

1 % The dictionary for this package.
2 %
3 %     \begin{macrocode}
4 /tx@LightThreeDDict 40 dict def
5 tx@LightThreeDDict begin
6 %     \end{macrocode}

```

⁵⁸For explanations of them, see [51] or other references on the PostScript programming language.

⁵⁹For instance, with a TeTeX distribution, it would be put logically in the TeX hierarchy local to the system, so in the /usr/local/share/texmf/dvips/pstricks subdirectory.

The LightThreeDMove operator, which will replace the moveto operator in the pathforall call. Syntax is: <x> <y> LightThreeDMove

```

1 %   The \texttt{LightThreeDMove} operator, which will replace the
2 %   \texttt{moveto} operator in the \texttt{pathforall} call.
3 % Syntax is: \Verb+<x> <y> LightThreeDMove+
4 %
5 %   \begin{macrocode}
/LightThreeDMove {
7 /y0c exch def
8 /x0c exch def
9 /xc x0c def
10 /yc y0c def
11 newpath
12 } def
13 %   \end{macrocode}
```

The LightThreeDLine operator, which will replace the lineto operator in the pathforall call. Syntax is: <x> <y> LightThreeDLine

```

1 %   The \texttt{LightThreeDLine} operator, which will replace the
2 %   \texttt{lineto} operator in the \texttt{pathforall} call.
3 % Syntax is: \Verb+<x> <y> LightThreeDLine+
4 %
5 %   \begin{macrocode}
/LightThreeDLine {
7 /yyc exch def
8 /xxc exch def
9 yyc yc sub
10 xxc xc sub
11 1 index 0 eq 1 index 0 eq and not {
12     atan
13     /ac exch def
14     ac LightThreeDMINangle le
15     ac LightThreeDMAngle ge or {
16         ac LightThreeDAngle sub
17         2 mul
18         cos
19         1 add
20         LightThreeDColorPsCommand
21         xc yc moveto
22         xxc yyc lineto
23         LightThreeDDXa LightThreeDDYa rlineto
24         xc LightThreeDDXa add yc LightThreeDDYa add lineto
25         closepath
26         fill
27     } if
28 } if
29 /xc xxc def
30 /yc yyc def
31 } def
```

```
32 % \end{macrocode}
```

The LightThreeDCurve operator, which will replace the curveto operator in the pathforall call. Syntax is:

```
<x2> <y2> <x3> <y3> <x4> <y4> LightThreeDCurve
```

```
1 % The \texttt{LightThreeDCurve} operator, which will replace the
2 % \texttt{curveto} operator in the \texttt{pathforall} call.
3 % Syntax is:
4 % \Verb+<x2> <y2> <x3> <y3> <x4> <y4> LightThreeDCurve+
5 %
6 % \begin{macrocode}
7 /LightThreeDCurve {
8   /y4c exch def
9   /x4c exch def
10  /y3c exch def
11  /x3c exch def
12  /y2c exch def
13  /x2c exch def
14  /y1c yc def
15  /x1c xc def
16 % \end{macrocode}
```

It is faster to make some preliminary computations outside the loop, as in the original file from Peter Kleiweg, but with T_EX it can generate huge numbers which make the PostScript interpreter crash!⁶⁰

```
1 % It is faster to make some preliminary computations outside
2 % the loop, as in the original file form Peter Kleiweg, but it
3 % can generate huge numbers which make the PostScript
4 % interpreter crash!\footnote{%
5 % It still can occur on some (unknown) circumstances. In such
6 % cases, just moving the text or graphic is generally enough
7 % to cure the problem...}
8 %
9 % \begin{macrocode}
10 % Computations order changed from the original file
11 1 LightThreeDSteps div 1 LightThreeDSteps div 1 {
12   /t exch def
13   3 t sub x1c mul
14   t 2 sub x2c mul
15   1 t sub x3c mul add 3 mul add
16   x4c t mul add t mul
17   x2c x1c sub 3 mul add t mul
18   x1c add% X
19   3 t sub y1c mul
20   t 2 sub y2c mul
```

⁶⁰It still can occur on some (unknown) circumstances. In such cases, just moving the text or graphic is generally enough to cure the problem...

```

21      1 t sub y3c mul add 3 mul add
22      y4c t mul add t mul
23      y2c y1c sub 3 mul add t mul
24      y1c add% Y
25      LightThreeDLine
26  } for
27 } def
28 %  \end{macrocode}

```

DG: Code modified because the original one crash the PostScript interpreter of two printers that I tested, even if GhostScript handle it correctly...

```

1  %  The \texttt{LightThreeDClose} operator, which will replace
2  %  the \texttt{closepath} operator in the \texttt{pathforall} call.
3  %  Syntax is: \Verb+LightThreeDClose+
4  %
5  %  \begin{macrocode}
6 /LightThreeDClose {
7  %  x0c y0c LightThreeDLine% Anomaly on some printers!
8  x0c 0 eq {x0c} {x0c 1 add} ifelse
9  y0c 0 eq {y0c} {y0c 1 add} ifelse LightThreeDLine
10 newpath
11 } def
12 %  \end{macrocode}

```

The LightThreeDPathForAll operator, which will replace the pathforall operator to draw the current path with a three dimensional lighten effect.

```

1  %  The \texttt{LightThreeDPathForAll} operator, which will
2  %  replace the \texttt{pathforall} operator to draw the current
3  %  path with a three dimensional lighten effect.
4  %
5  %  \begin{macrocode}
6 /LightThreeDPathForAll {
7  { LightThreeDMove} { LightThreeDLine }
8  { LightThreeDCurve } { LightThreeDClose } pathforall} def
9  %  \end{macrocode}

```

Then, to finish, we close the dictionary.

```

1  %  Then, to finish, we close the dictionary.
2  %
3  %  \begin{macrocode}
4 end
5 %  \end{macrocode}

```

```

1  %  \begin{macrocode}
2 %</postscript-header>
3 %  \end{macrocode}

```

And, at the end, we print the index and the list of changes of the package.

```
1 % \Finale
2 % \PrintIndex
3 % \PrintChanges
4 %
5 \endinput
6 %%
7 %% End of file 'pst-li3d.dtx'
```

Now, the new package can be used!

Test
Test

```
1 \DeclareFixedFont{\Bf}{T1}{ptm}{b}{n}{3cm}
2 \PstLightThreeDText[fillstyle=solid,fillcolor=white]{\Bf Test}
```

```
1 \PstLightThreeDText[linestyle=none,fillstyle=solid,
2 fillcolor=darkgray]{\Bf Test}
```

```
1 \psset{linestyle=none,fillstyle=solid,fillcolor=darkgray}%
2 \PstLightThreeDText[LightThreeDAngle=0]{\Bf Test}
3 \hfill
4 \PstLightThreeDText[LightThreeDAngle=90]{\Bf Test}
```

Test Test

```
1 \psset{linestyle=none,fillstyle=solid,fillcolor=darkgray}%
2 \PstLightThreeDText[LightThreeDXLength=0.5,
3 LightThreeDYLength=-1]{\Bf Test}
4 \hfill
5 \PstLightThreeDText[LightThreeDXLength=-1,
6 LightThreeDYLength=0.5]{\Bf Test}
```

Test Test

```

1 \DeclareFixedFont{\Sf}{T1}{phv}{b}{n}{3cm}
2 \psset{linestyle=none,fillstyle=solid,fillcolor=darkgray}%
3 \PstLightThreeDText[LightThreeDColorPsCommand=%
4   1.2 div setgray]{\Sf 123}
5 \hfill
6 \PstLightThreeDText[LightThreeDColorPsCommand=%
7   2.5 div setgray]{\Sf 123}

```



```

1 \DeclareFixedFont{\Rm}{T1}{ptm}{m}{n}{3cm}
2 \psset{linestyle=none,fillstyle=solid}%
3 \PstLightThreeDText[fillcolor=Violet,
4   LightThreeDColorPsCommand=%
5   2.5 div 0.7 exch 0.8 sethsbcolor]{%
6   \Rm 987}
7 \hfill
8 \PstLightThreeDText[fillcolor=DarkGreen,
9   LightThreeDColorPsCommand=%
10  2 div 0.5 exch 0.2 exch sethsbcolor]{%
11  \Rm 987}

```



```

1 \DeclareFixedFont{\Rmb}{T1}{ptm}{m}{n}{4cm}
2 \PstLightThreeDText[linestyle=none,fillstyle=solid,fillcolor=Gold,
3   LightThreeDColorPsCommand=%
4   1.2 div 0.15 exch 0.7 exch sethsbcolor]{%
5   \Rmb PSTricks}

```



```

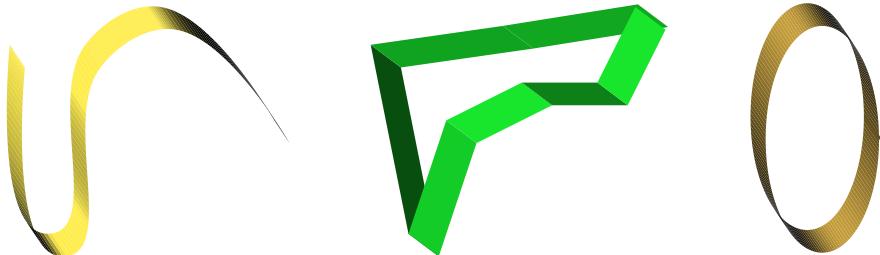
1 \psset{unit=0.5}%
2 \pspicture(-0.1,-3.5)(7.2,3)

```

```

3  \PstLightThreeDGraphic[LightThreeDXLength=0.4,
4      LightThreeDCOLORPsCommand=%
5          1.2 div 0.15 exch 0.7 exch sethsbcolor]{%
6          \pscurve(0,2)(1,-3)(2,2)(4,3)(7,0)}
7  \endpspicture
8  \hfill
9  \pspicture(0,-3.5)(7.7,3)
10 \PstLightThreeDGraphic[LightThreeDXLength=0.8,
11     LightThreeDCOLORPsCommand=%
12         2 div 0.35 exch 0.9 exch sethsbcolor]{%
13         \pspolygon(0,2)(1,-3)(2,0)(4,1)(6,1)(7,3)}
14 \endpspicture
15 \hfill
16 \pspicture(0.5,-3.6)(3.8,3)
17 \PstLightThreeDGraphic[LightThreeDCOLORPsCommand=%
18     2.6 div 0.12 exch 0.7 exch sethsbcolor]{%
19     \psellipse(2,0)(1.5,3)}
20 \endpspicture

```

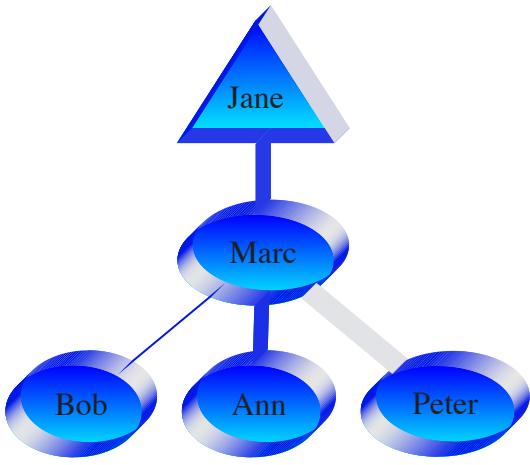


Here we draw an ellipse. The second case is more sophisticated, as we will increase continuously the starting value of the Brightness component of the HSB color, using a PostScript variable.

```

1 \SpecialCoor
2 \def\PstCoordinates{}%
3 \Multido{\nDistance=0.00+0.02,\iAngle=0+20}{200}{%
4     \edef\PstCoordinates{\PstCoordinates(\nDistance,\iAngle)}}
5 \psset{unit=0.5}%
6 \pspicture(-3.8,-4)(4.1,3.7)
7 \PstLightThreeDGraphic[LightThreeDLength=0.2,
8     LightThreeDCOLORPsCommand=%
9         1.2 div 0.3 exch 0.7 exch sethsbcolor]{%
10        \expandafter\pscurve\PstCoordinates}
11 \endpspicture
12 \hfill
13 \pspicture(-3.8,-4)(4.1,3.7)
14 \PstLightThreeDGraphic[LightThreeDLength=0.2,
15     LightThreeDColor=30,LightThreeDCOLORPsCommand=%
16         /Counter Counter 0.00005 add def
17         2 mul Counter exch 0.7 exch sethsbcolor]{%
18         \pstVerb{/Counter 0 def}%
19         \expandafter\pscurve\PstCoordinates}
20 \endpspicture

```

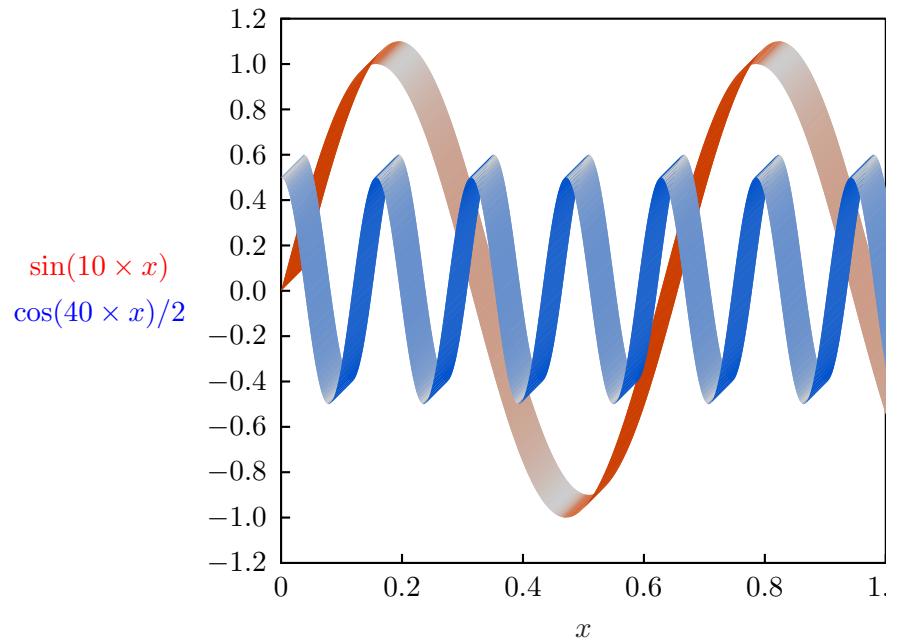


```

1 \PstLightThreeDGraphic[LightThreeDXLength=0.2,
2   LightThreeDYLength=-0.2,
3   LightThreeDCOLORPsCommand=%
4     1.2 div 0.65 exch 0.9 sethsbcolor]{%
5   \large
6   \let\Toval\ORIG\Toval
7   \def\Toval#1{\Toval\ORIG\raise2mm\hbox{\hskip2mm#1}}}{%
8   \let\Ttri\ORIG\Ttri
9   \def\Ttri#1{\Ttri\ORIG\raise3mm\hbox{#1}}}{%
10  \psset{framesep=0.15,fillstyle=gradient,gradmidpoint=0,
11    gradbegin=cyan,gradend=blue}}{%
12  \pstree[treesep=0.5]{\Ttri{Jane}}{%
13    \psset{framesep=0.25}}{%
14    \pstree{\Toval{Marc}}{%
15      \Toval{Bob}\Toval{Ann}\Toval{Peter}}}}}}}
```

```

1 \psset{xunit=8,yunit=3}%
2 \pspicture*(-0.45,-1.6)(1,1.3)
3   \psaxes[Dx=0.2,Oy=-1.2,Dy=0.2,tickstyle=top,
4     axesstyle=frame](0,-1.2)(1,1.2)
5   \rput(-0.3,0.1){\textcolor{red}{$\sin{(10 \times x)}$}}
6   \rput(-0.3,-0.1){\textcolor{blue}{$\cos{(40 \times x) / 2}$}}
7   \rput(0.5,-1.5){$x$}
8   \psset{plotpoints=500,LightThreeDXLength=0.3,
9     LightThreeDYLength=-0.3}}{%
10  \PstLightThreeDGraphic[LightThreeDCOLORPsCommand=%
11    1.5 div 0.05 exch 0.8 sethsbcolor]{%
12    \psplot{0}{1}{x 10 mul 57.296 mul sin}}}{%
13  \PstLightThreeDGraphic[LightThreeDCOLORPsCommand=%
14    1.5 div 0.6 exch 0.8 sethsbcolor]{%
15    \psplot{0}{1}{x 40 mul 57.296 mul cos 2 div}}}{%
16 \endpspicture}
```



XIII

Obsolete commands

8 Color commands

This also means that the command `\gray` (or `\red`, etc.) can be used much like `\rm` or `\tt`, as in

```
{\gray This stuff should be gray.}
```

The commands `\gray`, `\red`, etc. can be nested like the font commands as well. There are a few important ways in which the color commands differ from the font commands:

1. The color commands can be used in and out of math mode (there are no restrictions, other than proper TeX grouping).
2. The color commands affect whatever is in their scope (e.g., lines), not simply characters.
3. The scope of the color commands does not extend across pages.
4. The color commands are not as robust as font commands when used inside box macros. See page ?? for details. You can avoid most problems by explicitly grouping color commands (e.g., enclosing the scope in braces {}) whenever these are in the argument of another command.⁶¹

You can define or redefine additional colors and grayscales with the following commands. In each case, *numi* is a number between 0 and 1. Spaces are used as delimiters—don’t add any extraneous spaces in the arguments.

`\newgray{color}{num}`

num is the gray scale specification, to be set by PostScript’s `setgray` operator. 0 is black and 1 is white. For example:

```
1 \newgray{darkgray}{.25}
```

⁶¹However, this is not necessary with the PSTricks LR-box commands, except when `\psverbboxtrue` is in effect. See Section K.

`\newrgbcolor{color}{num1 num2 num3}`

num1 num2 num3 is a *red-green-blue* specification, to be set by PostScript's `setrgbcolor` operator. For example,

```
1 \newrgbcolor{green}{0 1 0}
```

`\newhsbcolor{color}{num1 num2 num3}`

num1 num2 num3 is an *hue-saturation-brightness* specification, to be set by PostScript's `sethsbcolor` operator. For example,

```
1 \newhsbcolor{mycolor}{.3 .7 .9}
```

`\newcmykcolor{color}{num1 num2 num3 num4}`

num1 num2 num3 num4 is a *cyan-magenta-yellow-black* specification, to be set by PostScript's `newcmykcolor` operator. For example,

```
1 \newcmykcolor{hercolor}{.5 1 0 .5}
```

For defining new colors, the *rgb* model is a sure thing. *hsb* is not recommended. *cmyk* is not supported by all Level 1 implementations of PostScript, although it is best for color printing. For more information on color models and color specifications, consult the *PostScript Language Reference Manual*, 3rd Edition (Red Book), and a color guide.

Driver notes: The command `\pstVerb` must be defined.

9 Put commands

This is old documentation, but these commands will continue to be supported.

There is also an obsolete command `\Lput` for putting labels next to node connections. The syntax is

```
1 \Lput{labelsep}[refpoint]{rotation}(pos){stuff}
```

It is a combination of `\Rput` and `\put`, equivalent to

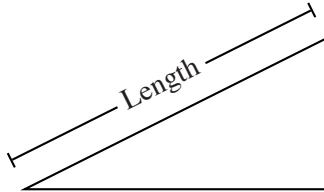
```
1 \put(pos){\Rput{labelsep}[refpoint]{rotation}(0,0){stuff}}
```

`\Mput` is a short version of `\Lput` with no `{<rotation>}` or `(<pos>)` argument. `\Lput` and `\Mput` remain part of PStricks only for backwards compatibility.

Here are the node label commands:

\lput*[refpoint]{rotation}(pos){stuff}

The **l** stands for “label”. Here is an example illustrating the use of the optional star and :<angle> with **\lput**, as well as the use of the **offset** parameter with **\pcline**:



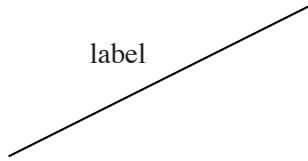
```

1 \pspolygon(0,0)(4,2)(4,0)
2 \pcline[offset=12pt]{l-l}(0,0)(4,2)
3 \lput*{:U}{Length}

```

(Remember that with the **put** commands, you can omit the coordinate if you include the angle of rotation. You are likely to use this feature with the node label commands.)

With **\lput** and **\rput**, you have a lot of control over the position of the label. E.g.,



```

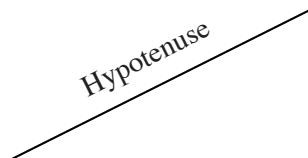
1 \pcline(0,0)(4,2)
2 \lput{:U}\rput[r]{N}(0,.4){label}

```

puts the label upright on the page, with right side located .4 centimeters “above” the position .5 of the node connection (above if the node connection points to the right). However, the **\lput** and **\bput** commands described below handle the most common cases without **\rput**.⁶²

\aput*[labelsep]{angle}(pos){stuff}

stuff is positioned distance **\pslabelsep** above the node connection, given the convention that node connections point to the right. **\aput** is a node-connection variant of **\lput**. For example:



```

1 \pspolygon(0,0)(4,2)(4,0)
2 \pcline[linestyle=none](0,0)(4,2)
3 \aput{:U}{Hypotenuse}

```

⁶²There is also an obsolete command **\Lput** for putting labels next to node connections. The syntax is

\Lput{labelsep}[refpoint]{rotation}(pos){stuff}

It is a combination of **\Rput** and **\lput**, equivalent to

\lput(pos)\Rput{labelsep}[refpoint]{rotation}(0,0){stuff}

\Mput is a short version of **\Lput** with no {<rotation>} or (<pos>) argument. **\Lput** and **\Mput** remain part of PStricks only for backwards compatibility.

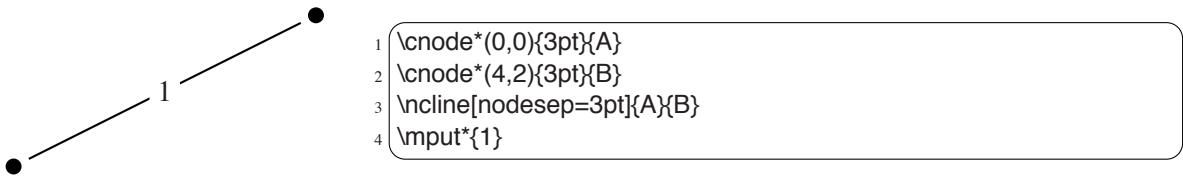
\bput*[labelsep]{angle}{(pos)}{stuff}

This is like **\aput**, but *stuff* is positioned *below* the node connection.

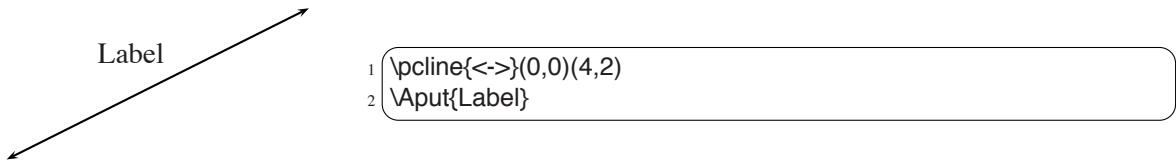
It is fairly common to want to use the default position and rotation with these node connections, but you have to include at least one of these arguments. Therefore, PSTricks contains some variants:

\mput*[refpoint]{stuff}
\Aput*[labelsep]{stuff}
\Bput*[labelsep]{stuff}

of **\put**, **\aput** and **\bput**, respectively, that have no angle or positioning argument. For example:



Here is another:



10 Coordinates commands

There is an obsolete command **\Polar** that causes coordinates in the form (r,θ) to be interpreted as polar coordinates. The use of **\Polar** is not recommended because it does not allow one to mix Cartesian and polar coordinates the way **\SpecialCoor** does, and because it is not as apparent when examining an input file whether, e.g., $(3,2)$ is a Cartesian or polar coordinate. The command for undoing **\Polar** was **\Cartesian**. It has an optional argument for setting the default units. I.e.,

```
1 \Cartesian(x,y)
```

has the effect of

```
1 \psset{xunit=x,yunit=y}
```

\Cartesian can be used for this purpose without using **\Polar**.

Help⁶³

We try to give in this appendix various hints about both common requirements (PDF output files, animated graphics, etc.), frequently asked questions (FAQ) and various programming technics of common interest.

In this last area, some (at least basic) knowledge of both \TeX and PostScript as programming languages are required, as PSTricks used both. This is outside the scope of this *User's Guide* to explain them, and their basis will be supposed to be known.

To learn about \TeX as a programming language, there is not currently a specific book or publication on this specific topic. Nevertheless, there are chapters on this subject in some general books about \TeX . In a more pedagogical way than in the *\TeXbook* [37], you can consult for instance *\TeX by Topic*, by Victor Eijkhout [12], or *The Advanced \TeXbook*, by David Salomon [77].

To learn about the PostScript language, consult Adobe's *PostScript Language Tutorial and Cookbook* (the “Blue Book”) [50], or for instance Henry McGilton and Mary Campione's *PostScript by Example* [39], or of course the *PostScript Language Reference Manual* [51]. You may find that the Appendix of the Blue Book, plus an understanding of how the stack works, is all you need to write simple code for computing numbers (e.g., to specify coordinates or plots using PostScript).

A PDF output files

DG: This chapter is not yet very stable and its English has not yet been corrected.

The *Portable Document Format* (PDF) [46], a proprietary but public format from Adobe Inc., became these last years well spread and more and more used. Nevertheless, this format does not supercede PostScript, heavily used by PSTricks, and is a description language which does not integrate programming capacities that PostScript have and that PSTricks use.

So, there is not an obvious unique way to generate PDF files with PSTricks graphics, but several ones, with various advantages and disadvantages each. Consequently, everyone must make his own choice among the various solutions available.

⁶³This chapter was rewritten in 2003 by Denis Girou.

A.1 Compilers with internal support of PStricks

Two currently available compilers, Bakoma \TeX [5] from Basil K. Malyshев and VT \TeX [82] from MicroPress Inc., among those which can directly produce PDF output files, embed a PostScript interpreter (at least all the major parts of the defined language, which allow them to include directly EPS external files and PStricks graphics). These two softwares are both commercial products for Windows systems, but MicroPress also distribute a free version of VT \TeX for Linux and some other systems.

A.2 PostScript to PDF converters

The Acrobat software [1] from Adobe, GhostScript [20] from Aladdin Enterprises and PStill [66] from Frank Siegert allow to translate PostScript files to PDF. The first one exist only for Macintosh and Windows systems and the two other ones for all the major systems. Acrobat is a paying product, GhostScript is a free software and PStill is free for academic usage, and commercial otherwise.

A.3 Pdf \TeX workarounds

The Pdf \TeX compiler [47] does not embed a PostScript interpreter, and consequently cannot include EPS external files without previous conversion of them in another accepted format (which could be done *on the fly*), nor directly PStricks graphics.

Consequently, if you want to use Pdf \TeX with PStricks, you must convert the PStricks graphics, manually or as an automatic process. Manually, you can store each PStricks graphic in an external EPS file using either the **\PSTtoEPS** macro or the **TeXtoEPS** environment (see section 61), convert these files to another format supported by Pdf \TeX then include these files in the final document.

More easily, you could use either the pdfTricks system [48] developed by Radhakrishnan CV, Rajagopal CV and Antoine Chambert-Loir, or the ps4pdf system [53] developed by Rolf Niepraschk, which both offer a way to automatize this process. They both require to encapsulate the PStricks graphics in a special environment or macro (and work only for L \TeX , not PLAIN \TeX).

pdfTricks require a \TeX implementation based on **web2c**. It is easier to use if you enable the so-called *shell escape* feature of the compiler, to allow the execution of external processes during the \TeX compilation,⁶⁴ but it can

⁶⁴For security reasons, this is not recommended for computers accessible from outside by the Internet network to have this feature permanent, but better as a temporary opened feature during the time of the compilations.

work in another way without using this feature, using a Shell script given in the distribution.

Here is a simple example of usage:

```

1 \documentclass{article}
2
3 \usepackage{pdftricks}
4
5 \begin{psinputs}
6   \usepackage{times}
7   \usepackage{pstcol}
8   \usepackage{pst-plot}
9 \end{psinputs}
10
11 \begin{document}
12
13 \begin{pdfpic}
14   \psset{xunit=8,yunit=3}%
15   \begin{pspicture}(-0.45,-1.6)(1.05,1.3)
16     \psaxes[Dx=0.2,Oy=-1.2,Dy=0.2,tickstyle=top,
17               axesstyle=frame](0,-1.2)(1,1.2)
18   \rput(-0.3,0){%
19     \shortstack{\textcolor{red}{$\sin(10 \times x)$}\ \
20               \textcolor{blue}{$\cos(40 \times x) / 2$}}}
21   \rput(-0.3,0){%
22     \shortstack{\textcolor{red}{$\sin(10 \times x)$}\ \
23               \textcolor{blue}{$\cos(40 \times x) / 2$}}}
24   \rput(0.5,-1.5){\shortstack{$x$\\Sample plots}}
25   \psplot[linecolor=red]{0}{1}{x mul 57.296 mul sin}%
26   \psplot[linecolor=blue,plotpoints=200]{0}{1}{x 40 mul 57.296 mul cos 2 div}%
27   \end{pspicture}
28 \end{pdfpic}
29
30 \end{document}
31
32 \end{document}
```

which must produce the following result:

The command to compile this file, and directly produce the PDF output file, is the following (if the *shell escape* feature is not enabling by default):

```

1 pdflatex -shell-escape FileName
```

On its side, ps4pdf require a PostScript to PDF converter like GhostScript. It also needs the ‘preview’ package [52] from David Kastrup. With ps4pdf,

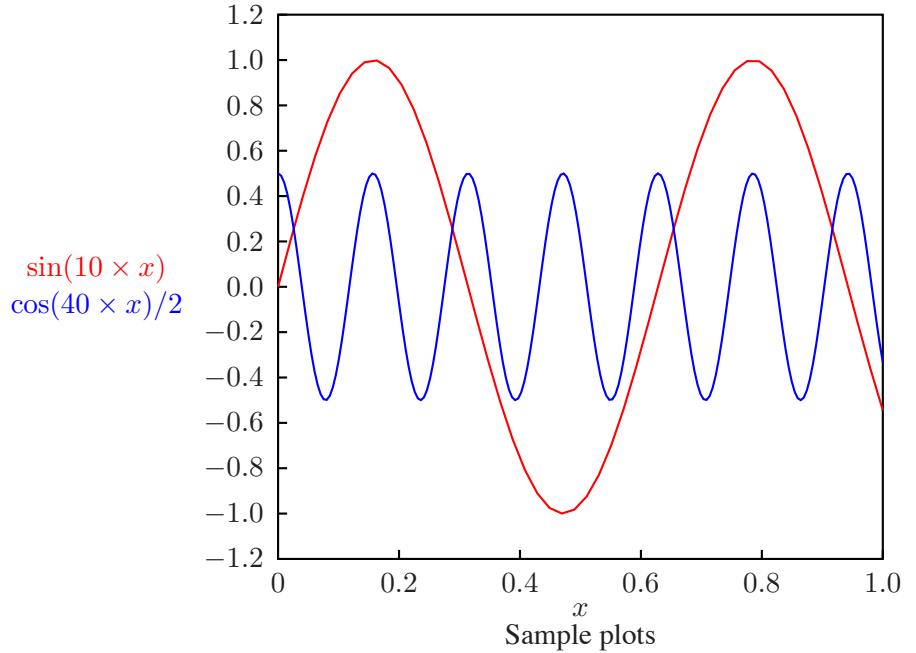


Figure 4: Output of the example about pdfTricks and ps4pdf usage

the preceding example must look like:

```

1 \documentclass{article}
2
3 \usepackage{graphicx}
4 \usepackage{ps4pdf}
5
6 \PSforPDF{%
7   \usepackage{pstcol}
8   \usepackage{pst-plot}}
9
10 \begin{document}
11
12 \PSforPDF{%
13   \psset{xunit=8,yunit=3}%
14   \begin{pspicture}(-0.45,-1.6)(1.05,1.3)
15     \psaxes[Dx=0.2,Oy=-1.2,Dy=0.2,tickstyle=top,
16             axesstyle=frame](0,-1.2)(1,1.2)
17     \rput(-0.3,0){%
18       \shortstack{\textcolor{red}{$\sin (10 \times x)$}\\"\\
19           \textcolor{blue}{$\cos (40 \times x) / 2$}}}
20     \rput(0.5,-1.5){\shortstack{$x$\Sample plots}}
21     \psplot[linecolor=red]{ 1 radian = 57.296 degrees
22     {0}{1}{x 10 mul 57.296 mul sin} sin(10 x)
23     \psplot[linecolor=blue,plotpoints=200]

```

```

24      {0}{1}{x 40 mul 57.296 mul cos 2 div}%  $\cos(40 x) / 2$ 
25      \end{pspicture}}
26
27 \end{document}

```

And the following sequence of commands must be given to generate the PDF output file (this is of course easier to have a Makefile or a script to do that automatically):

```

1 latex FileName
2 dvips -Ppdf -G0 -o FileName-pics.ps FileName.dvi
3 ps2pdf FileName-pics.ps
4 pdflatex FileName

```

B Animated graphics

Sometimes, graphics are interesting to show the evolution of a phenomena. Of course, it is possible to include in a \TeX document several pictures of the phenomena, at different steps, but it could be more convenient to generate real animations to be viewed interactively, using one of the available animated graphics file formats (mainly GIF and MNG (*Multiple-image Network Graphics*) [44].)

Two different cases must be considered: the one of *pure* PSTRicks graphics and the one of *non pure* graphics.

B.1 Pure PSTRicks graphics

This first case (without texts, `\rput` macros, etc.) is the easiest, as all the images of the animation can be generated in only one compilation, using a loop at the \TeX level on the `\PSTtoEPS` macro (see section 61), which can dynamically transform each PSTRicks graphic computed in an EPS file.

We will give an example on waves interferences, a simple but powerful development done by Manuel Luque in 2002. The purpose of this code is to simulate the behavior of one or two waves, and the interferences between them when there are two ones. The physical phenomena of interferences come from the superposition of two vibrating movements generated by two sources which have a constant difference of phase.

The code only consist in few PostScript lines, to show the propagation of waves from one source or the interferences between the waves of two sources.

There are many parameters which can be changed:

Time Time of the observation (real, default=0)
X1 Abscissa of the first source, in centimeters (real, default=-2)
X2 Abscissa of the second source, in centimeters (real, default=2)
Y1 Ordinate of the first source, in centimeters (real, default=0)
Y2 Ordinate of the second source, in centimeters (real, default=0)
Phase1 Phase of the first source, in degrees (real, default=0)
Phase2 Phase of the second source, in degrees (real, default=0)
Frequency1 Frequency of the first source, in Hz (real, default=20)
Frequency2 Frequency of the second source, in Hz (real, default=20)
Amplitude1Coeff Amplitude coefficient first source; a value of 0 mean no
first source (real between 0 and 1, default=1)
Amplitude2Coeff Amplitude coefficient for the second source; a value of
0 mean no second source (real between 0 and 1, default=1)
Amortization Amortization factor (real, default=5)
Celerity Speed of the waves, in ms^{-1} (real, default=0.2)
Resolution Scale factor for the image (1 is the more precise resolution)
(integer, default=1)

Here, we will generate only three images, but it is obviously straightforward to generate as many as we want. We must just take care to generate the image number with the correct left leading zeroes (in our example, we show how to do to generate convenient name files until 99 ones.)

```

1 \define@key{psset}{Amortization}{%
2   \edef\PstWavesInterferences@Amortization{\#1}}
3 \define@key{psset}{Amplitude1Coeff}{%
4   \edef\PstWavesInterferences@AmplitudeACoeff{\#1}}
5 \define@key{psset}{Amplitude2Coeff}{%
6   \edef\PstWavesInterferences@AmplitudeBCoeff{\#1}}
7 \define@key{psset}{Celerity}{%
8   \edef\PstWavesInterferences@Celerity{\#1}}
9 \define@key{psset}{Frequency1}{%
10  \edef\PstWavesInterferences@FrequencyA{\#1}}
11 \define@key{psset}{Frequency2}{%
12  \edef\PstWavesInterferences@FrequencyB{\#1}}
13 \define@key{psset}{Phase1}{%
14  \edef\PstWavesInterferences@PhaseA{\#1}}
15 \define@key{psset}{Phase2}{%
16  \edef\PstWavesInterferences@PhaseB{\#1}}
17 \define@key{psset}{Resolution}{%
18  \edef\PstWavesInterferences@Resolution{\#1}}
19 \define@key{psset}{Time}{\edef\PstWavesInterferences@Time{\#1}}
20 \define@key{psset}{X1}{%
21  \edef\PstWavesInterferences@SourceXA{\#1}}
22 \define@key{psset}{X2}{%
23  \edef\PstWavesInterferences@SourceXB{\#1}}
24 \define@key{psset}{Y1}{%
25  \edef\PstWavesInterferences@SourceYA{\#1}}

```

```

26 \define@key{psset}{Y2}{%
27   \edef\PstWavesInterferences@SourceYB{\#1}
28
29 \setkeys{psset}{Amortization=5,Amplitude1Coeff=1,Amplitude2Coeff=1,
30   Celerity=0.20,Frequency1=20,Frequency2=20,Phase1=0,Phase2=0,
31   Resolution=1,Time=0.20,X1=-2,X2=2,Y1=0,Y2=0}
32
33 \def\PstWavesInterferences{%
34   \def\pst@par{}\pst@object{\PstWavesInterferences}}
35
36 \def\PstWavesInterferences@i(#1,#2)(#3,#4){%
37   \pspicture(#1,#2)(#3,#4)
38   \begin{AltOpenObj}
39     \addto@pscode{%
40       /AmplitudeMax 0.5 def
41       #1 28.45 mul \PstWavesInterferences@Resolution\space
42         #3 28.45 mul { % Loop on abscissas
43           /abscissept exch def % In points
44           /abscisse abscissept 2845 div def % In meters
45           #2 28.45 mul \PstWavesInterferences@Resolution\space
46             #4 28.45 mul { % Loop on ordinates
47               /ordonneept exch def % In points
48               /ordonnee ordonneept 2845 div def % In meters
49               /d1 abscisse \PstWavesInterferences@SourceXA\space
50                 100 div sub dup mul
51                 ordonnee \PstWavesInterferences@SourceYA\space
52                 100 div sub dup mul add sqrt def
53               /d2 abscisse \PstWavesInterferences@SourceXB\space
54                 100 div sub dup mul
55                 ordonnee \PstWavesInterferences@SourceYB\space
56                 100 div sub dup mul add sqrt def
57               /yS1 360 \PstWavesInterferences@FrequencyA\space mul
58                 \PstWavesInterferences@Time\space d1
59                 \PstWavesInterferences@Celerity\space div sub mul
60                 \PstWavesInterferences@PhaseA\space add sin
61                 AmplitudeMax 2.71828 d1 neg
62                 \PstWavesInterferences@Amortization\space mul exp mul
63                 \PstWavesInterferences@AmplitudeACoeff\space mul mul def
64               /yS2 360 \PstWavesInterferences@FrequencyB\space mul
65                 \PstWavesInterferences@Time\space d2
66                 \PstWavesInterferences@Celerity\space div sub mul
67                 \PstWavesInterferences@PhaseB\space add sin
68                 AmplitudeMax 2.71828 d2 neg
69                 \PstWavesInterferences@Amortization\space mul exp mul
70                 \PstWavesInterferences@AmplitudeBCoeff\space mul mul def
71               yS1 0.5 add yS2 0.5 add 1 setrgbcolor % Color
72               newpath
73               abscissept ordonneept
74                 \PstWavesInterferences@Resolution\space 0 360 arc
75               closepath
76               fill
77             } for
78           } for}%

```

```

79 \end{OpenObj}
80 \endpspicture}
81
82 % Generation of the images in EPS format
83 \multido{\nTime=0.20+0.01}{3}{%
84   % As all file names must have the same number of characters,
85   % we must add a 0 for the 9 first ones if we have between
86   % 10 and 99 images
87   \ifnum\multidocount<10\relax\def\Pad{0}\else\def\Pad{}\fi
88   \PSTtoEPS[headerfile=pstricks.pro,headers=user,
89             bbllx=-8,bbly=-4,bburx=8,bbury=4]
90             {WavesInterferences}\Pad\the\multidocount.eps\%
91   \PstWavesInterferences[Amortization=10,Celerity=0.2,
92                         Time=\nTime,Frequency1=20,Frequency2=20,Resolution=3,
93                         X1=-4,X2=4,Y1=-3,Y2=-3](-8,-4)(8,4)}}

```

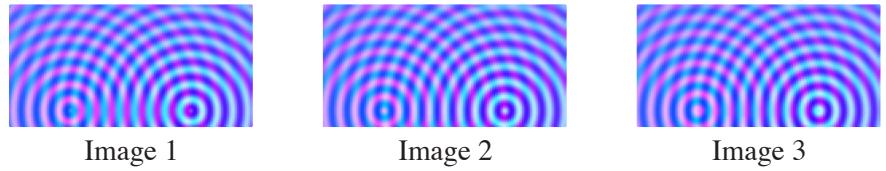


Image 1

Image 2

Image 3

Now, to generate the animated graphic file which concatenate these images, we have to convert each file in a convenient format, then to use a tool to concatenate them. For instance, the `convert` program from ImageMagick [32] (or the `gm` one from GraphicsMagick [26]) can do all of these tasks, both for GIF and MNG formats. Nevertheless, take care that sometimes it could generate huge files. For GIF files, the `gifsicle` [21] application usually produce smaller files.

Here we give an example script for Unix systems which allow to automatize all the process.

```

1 #! /bin/sh
2
3 set -x
4
5 # Generation of an animated GIF file
6 # from PSTricks pure graphics
7
8 if [ $# != 2 ]
9 then echo Syntax: PstAnimatedPureGraphics.sh \
10           file_without_suffix delay
11         exit
12 fi
13
14 latex $1
15
16 for file1 in $1?*.eps

```

```

17 do
18   file2='basename $file1 .eps'
19   convert $file1 $file2.gif # Convert each file in GIF format
20 done
21
22 # Generate the animated GIF file using "gifsicle"
23 gifsicle -delay $2 -loop -colors 256 -output $1.gif $1?*.gif
24
25 # We can also use "convert" from ImageMagick
26 # (but it could produce in some cases huge files)
27 # convert -delay $2 -loop 0 -colors 256 $1?*.gif $1.gif
28
29 # Clean up temporary files
30 rm -f $1?*.{eps,gif} $1.{aux,dvi,log}
31
32 animate $1.gif & # From ImageMagick, or "gifview -a $1.gif"
33
34 exit

```

If this script is named `PstAnimatedPureGraphics.sh` and our `\TeX` file `WavesInterferences.tex`, we must just execute the following line (the last number is the delay between images, in cent of second):

```
1 PstAnimatedPureGraphics.sh WavesInterferences 20
```

B.2 Non pure **PSTricks** graphics

This case is a little more complicated, because each EPS file must be generated independently by a separate compilation, using the `\TeXtoEPS` environment and a translation from the DVI format to the EPS one (see section 61). Nevertheless, everything can also be automatized using a script.

We give here an example based on the ‘`pst-eucl`’ contribution package [59] from Dominique Rodriguez, for Euclidean geometry drawings.

```

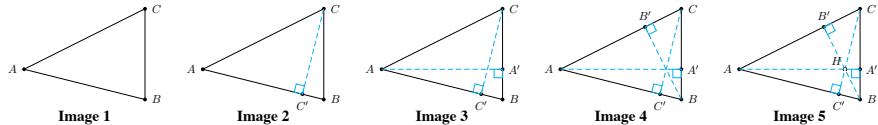
1 \documentclass{minimal}
2
3 \usepackage{pst-eps}
4 \usepackage{pst-eucl}
5
6 % Utility macro to generate intermediate images,
7 % running parts of the complete code of a macro.
8 % Code #3 will be included if #1 >= #2.
9 \def\IncludeCodeForImage#1#2#3{%
10 \ifnum#2>#1\relax
11 \else

```

```

12 #3
13 \fi}
14
15 \def\PstOrthocentre{%
16 \typein[\Image]{^^JWhich image?}
17 \TeXtoEPS
18 \pspicture(-1.5,-1.3)(3.6,2.2)
19 \psset{CodeFig=true}%
20 \IncludeCodeForImage{\Image}{1}%
21 \pstTriangle[PosAngleA=180](-1,0){A}(3,-1){B}(3,2){C}
22 \IncludeCodeForImage{\Image}{2}%
23 \pstProjection[PosAngle=-90]{B}{A}{C}{C'}
24 \IncludeCodeForImage{\Image}{3}%
25 \pstProjection{B}{C}{A}{A'}
26 \IncludeCodeForImage{\Image}{4}%
27 \pstProjection[PosAngle=90]{A}{C}{B}{B'}
28 \IncludeCodeForImage{\Image}{5}%
29 \pstInterLL[PosAngle=135,PointSymbol=square]
30 {A}{A'}{B}{B'}{H}
31 \endpspicture
32 \endTeXtoEPS}
33
34 \begin{document}
35 \PstOrthocentre
36 \end{document}

```



Here we give an example script for Unix systems which allow to automatize all the process.

```

1 #! /bin/sh
2
3 set -x
4
5 # Generation of an animated GIF file
6 # from PSTricks non pure graphics
7
8 if [ $# != 3 ]
9 then echo Syntax: PstAnimatedNonPureGraphics.sh \
10      file_without_suffix number_images delay
11      exit
12
13 declare -i i

```

```

14
15 i=1
16 while [ $i -le $2 ]
17 do
18   echo $i | latex Orthocentre
19   if [ $i -lt 10 ]
20     then pad="00"
21   else if [ $i -lt 100 ]
22     then pad="0"
23   else pad=""
24   fi
25 fi
26 dvips -E -o $1$pad$i.eps $1.dvi
27 # Convert the file in GIF format
28 convert $1$pad$i.eps $1$pad$i.gif
29 i=$i+1
30 done
31
32 # Generate the animated GIF file using "gifsicle"
33 gifsicle --delay 200 --loop --colors 256 \
34   --output $1.gif $1?*.gif
35
36 # We can also use "convert" from ImageMagick
37 # (but it could produce in some cases huge files)
38 # convert -delay 200 -loop 0 -colors 256 $1?*.gif $1.gif
39
40 # Clean up temporary files
41 rm -f $1?*.{eps,gif} $1.{aux,dvi,log}
42
43 animate $1.gif & # From ImageMagick, or "gifview -a $1.gif"
44
45 exit

```

If this script is named `PstAnimatedNonPureGraphics.sh` and our \TeX file `Orthocentre.tex`, we must just execute the following line (the first number is the number of images to generate and the second one is the delay between images, in cent of second):

<code>1 PstAnimatedNonPureGraphics.sh Orthocentre 5 200</code>
--

C Generation of code by external programs

Several drawing tools (Dia [10], Eukleides [14], gnuplot [24], jPicEdt [34], etc.) can generate PSTricks output code, that can be later inserted in any \TeX documents.

But you can of course write yourself some programs in any scripting or programming language to generate PSTricks output code for complex tasks.

Here is a rather simple script, written in Perl, which allow to draw disk file trees of Unix systems, analyzing the subdirectories. The script is rather short, but nevertheless powerful.

```
1  #! /usr/bin/perl
2
3  use Getopt::Long;
4
5  $TreeDirectory = ".";# Default directory is the current one
6  $RecursionLevelMax = 10;# Better than infinity!
7
8  &OptionsRead;
9  if (! -d $TreeDirectory) {
10    print "$TreeDirectory is not a directory!\n";
11    exit 0
12  }
13  print "\\\$tree{\\\$TreeDirectory{$TreeDirectory}}\\n {";
14  &ListSubdirectories($TreeDirectory, 0);
15  print "\\n";
16  exit 0;
17
18  # To get list of subdirectories (recursive subprogram)
19  sub ListSubdirectories {
20    local ($directory, $recursion_level) = @_;
21    local ($boolean, $dir_temp, $dir_print_temp, @dir, @subdir,
22          *DIR);
23
24    opendir (DIR, $directory) or return;
25    @dir = grep {!/\^\.\.?$/} grep { -d } map {"$directory/$_"}
26              readdir (DIR);
27    closedir (DIR);
28
29    $recursion_level++;
30
31    if ($recursion_level <= $RecursionLevelMax) {
32      $boolean = 0;
33      foreach $dir_temp (sort @dir) {# Loop on subdirectories
34        # For special characters that are a problem for TeX
35        $dir_print_temp = $dir_temp;
36        $dir_print_temp =~ s/.*/\\/, s/[\\$,\\&,\\%,\\#,\\_,\\{,\\}]\\/_/g;
37
38        if ($boolean != 0) {print "\\n", " " x (2*$recursion_level)}
39
40        # We must see if this directory has subdirectories
```

```

41     opendir (DIR, "$dir_temp");
42     @subdir = grep {!(/^\.\?$/)} grep { -d }
43             map {"$dir_temp/$_" } readdir (DIR);
44     closedir (DIR);
45
46     if ($#subdir != -1 &&
47         $recursion_level != $RecursionLevelMax) {
48         # At least one subdirectory
49         print "\\\pstree{\\\PstTreeDirectory{$dir_print_temp}}\\n",
50             " " x (2 * $recursion_level + 1), "{";
51     }
52     else {
53         print "\\\PstTreeDirectory{$dir_print_temp}";
54     }
55
56     $boolean = 1;
57     &ListSubdirectories("$dir_temp", $recursion_level);
58 }
59
60     if ($boolean != 0) {print "}";
61 }
62 }
63
64 # Reading options
65 sub OptionsRead{
66     GetOptions (qw (-d=s -rl=i));# Option definitions
67
68     # Option analysis
69     if ($opt_d) {$TreeDirectory = $opt_d};
70     if ($opt_rl) {$RecursionLevelMax = $opt_rl};
71 }
```

and here are some examples of usage, with various customizations. The commands given which have generated the files included are shown in the comments.

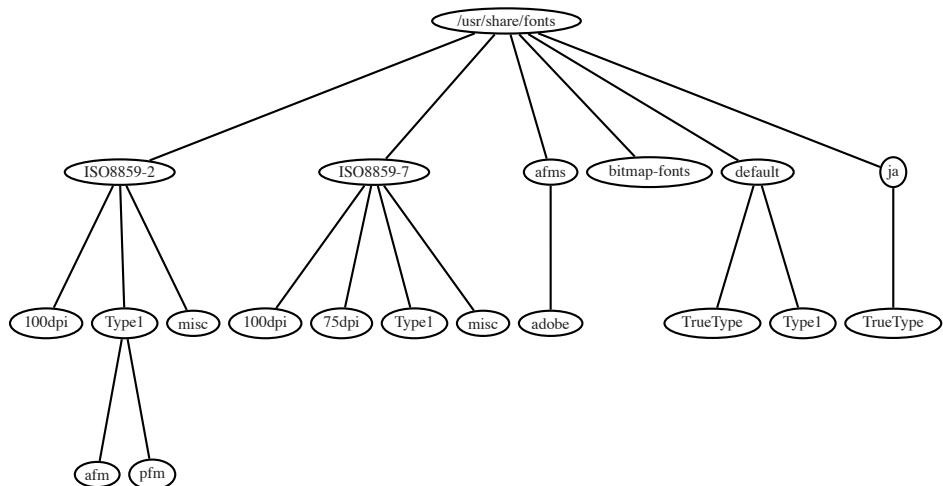
```

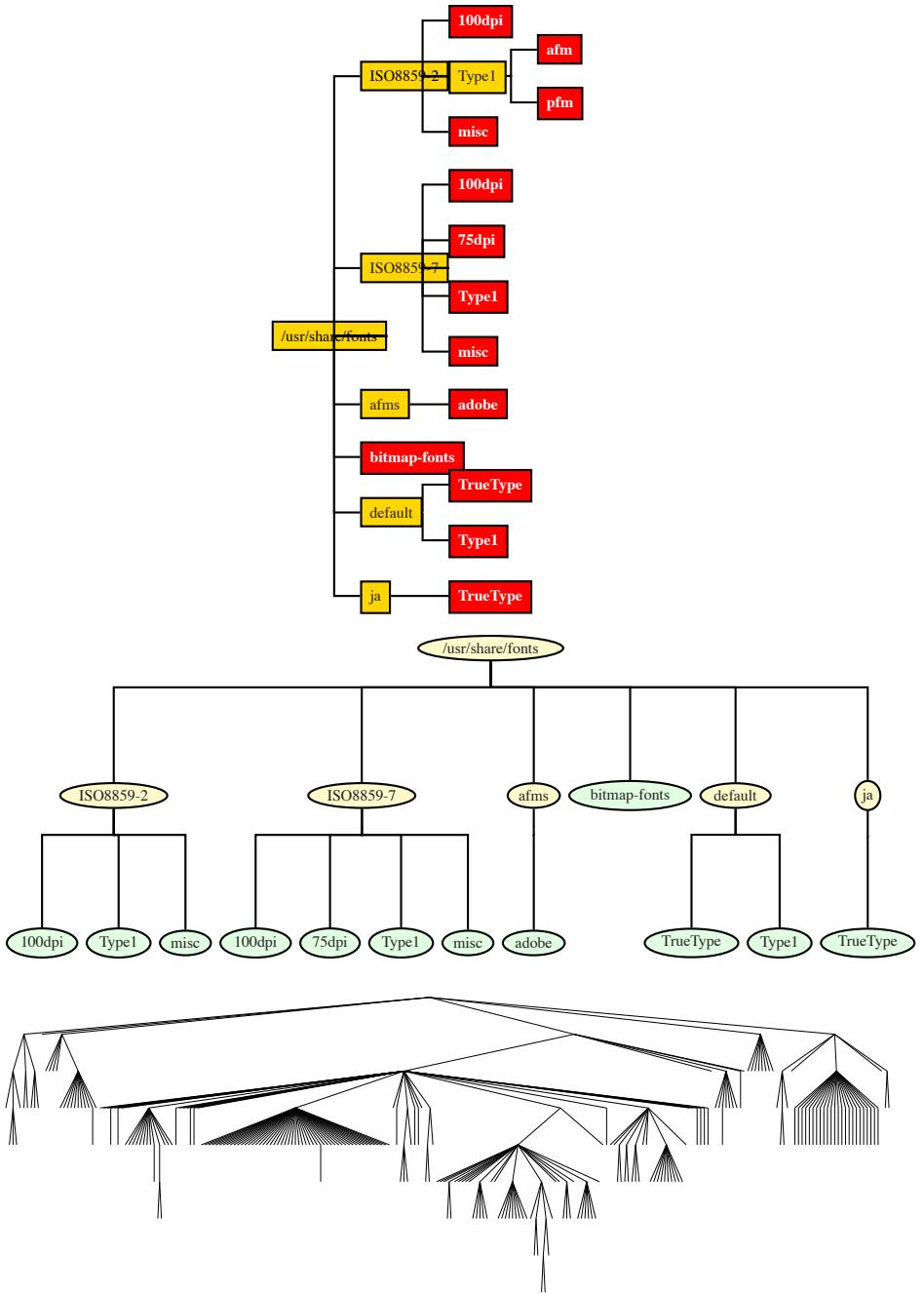
1 \tiny
2
3 % pst-dir.pl -d /usr/share/fonts >dirtree1.tex
4 \bgroup
5   \def\PstTreeDirectory#1{\Toval[framesep=0.05]{#1}%
6   \psset{treesep=0.1}%
7   \input{dirtree1.tex}
8 \egroup
9
10 \bgroup
11   \def\PstTreeDirectory#1{%
12     \ifx\pstree@next\relax
```

```

13   \def\pst@tempa{\psframebox[fillstyle=solid,fillcolor=Gold]{#1}}%
14 \else
15   \def\pst@tempa{\psframebox[fillstyle=solid,fillcolor=red]{%
16     \textcolor{white}{\textbf{#1}}}}%
17 \fi
18 \TR{\pst@tempa}%
19 \def\psedge{\ncangle[angleB=180]}% Horizontal edges
20 \psset{treemode=R,treesep=0.3,href=-1,levelsep=*1.2}%
21 \input{dirtree1.tex}
22 \egroup
23
24 % pst-dir.pl -d /usr/share/fonts -rl 2 >dirtree2.tex
25 \bgroup
26 \psset{treesep=0.1}%
27 \def\PstTreeDirectory#1{%
28 \ifx\pstree@next\relax
29   \def\pst@tempa{LemonChiffon}%
30 \else
31   \def\pst@tempa{PaleGreen}%
32 \fi
33 \Toval[framesep=0.05,fillstyle=solid,fillcolor=\pst@tempa]{#1}}%
34 \def\psedge{\ncangles[angleA=-90,angleB=90]}% Horizontal edges
35 \input{dirtree2.tex}
36 \egroup
37
38 % pst-dir.pl -d /usr/X11R6 >dirtree3.tex
39 \bgroup
40 \psset{treefit=loose,treesep=0.2}%
41 \def\PstTreeDirectory#1{\Tp}%
42 \scalebox{0.125}{\input{dirtree3.tex}}
43 \egroup

```





D Loops

Loops are a key basic construction in all programs and are often required also to build computational graphics. The macros **\multirput**, to place multiple copies of texts or graphics at equally spaced positions expressed in Cartesian coordinates, and **\multipsp** for pure PSTricks graphics placed at equally spaced positions, but not obligatory expressed by Cartesian coordinates, have been described in Section 25.

Examples of usage of the more general ‘multido’ package⁶⁵ [45] had already been shown in various preceding chapters, but this is important to emphasize its various features, as its usage is common.⁶⁶

Refer to its documentation for a complete description, but in few words:

- loops can be nested (the **\multido** and **\mmultido** ones, not the **\Multido** and **\MMultido** variant ones),
- a same loop can define several counters of different kinds, incremented together,
- increments can be either positive or negative,
- counters can have the *integer, real, number* (for integer or real numbers, but with the same number of digits for the decimal parts of starting value and increment)⁶⁷ or *dimension* types,
- the **\multido** macro puts its body inside a group, at the opposite of **\Multido**,
- the **\mmultido** and **\MMultido** variants increment the counters before starting,
- the **\multidostop** macro allows to interrupt a loop,
- the **\multidocount** counter stores the value of the iteration number.

We give here some other examples, to illustrate the various usages of such loops. You will still find other ones in the next sections of this chapter (specially look in the Section G the examples of how to define loops of undefined lengths and how to stop them with the **\multidostop** macro when a predefined condition is reached).

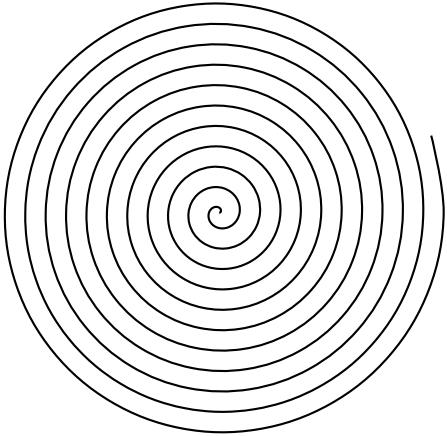
First, a simple example to draw a spiral. Note that:

- the straightforward way to draw a spiral is to use polar plots (see Section L.15),
- the technique used here to compute coordinates, to store them and reuse them later is very general,
- we use here the **\Multido** variant; if we would have used the **\multido** one, we must have used the **\xdef** macro to store the coordinates and not the **\edef** one, to be able to use it value outside the group automatically defined in the body of a **\multido** loop.

⁶⁵You can also use the ‘repeat’ package [74] from Victor Eijkhout or, if you use only L^AT_EX, the **\whiledo** macro of the ‘ifthen’ package [31] or the internal L^AT_EX macros **\@for**, **\@whilenum**, etc.

⁶⁶Nevertheless, in the restricted cases where it is applicable, the **\multip** and **\multirput** macros are more efficient and produce more compact code than the **\multido** one.

⁶⁷Usage of *reals* rather than *numbers* is faster but less precise, with possible rounding approximations during computations.



```

1 \SpecialCoor
2 \psset{unit=0.75}%
3 \def\PstCoordinates{}%
4 \Multido{\nDistance=0.00+0.02,\iAngle=0+20}{200}{%
5   \edef\PstCoordinates{\PstCoordinates(\nDistance;\iAngle)}}
6 \expandafter\pscurve\PstCoordinates

```

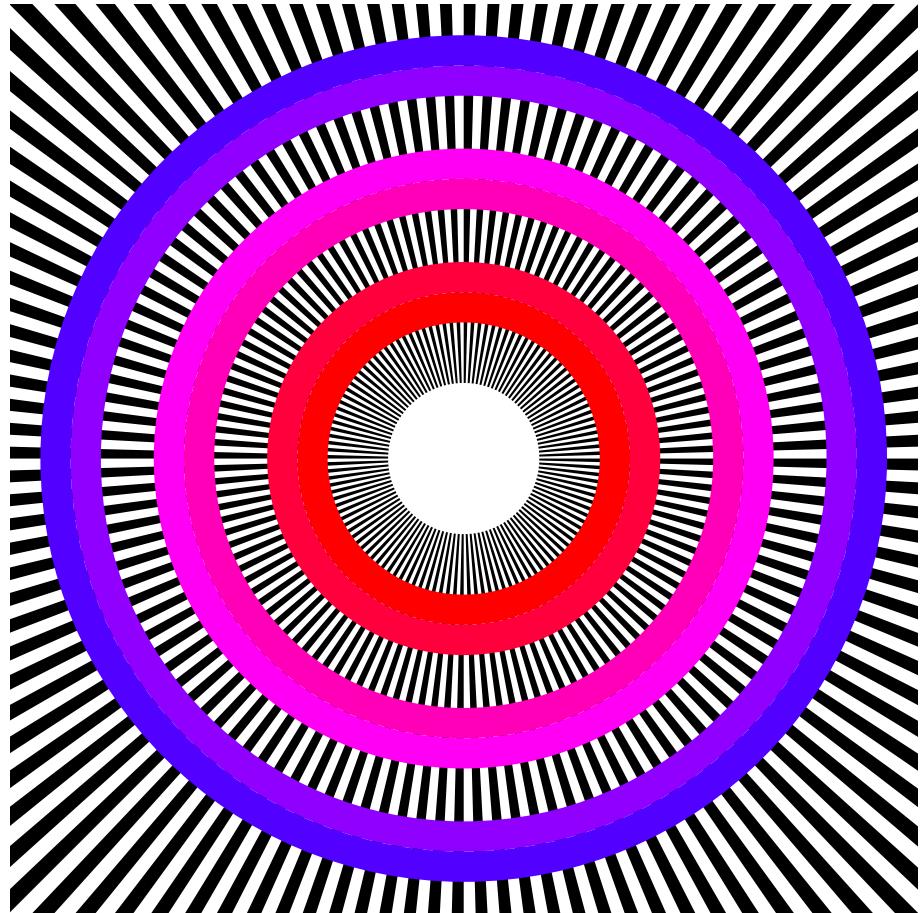
Then we rebuild the *Enigma* painting of the French painter Isia Leviant.⁶⁸ Note the usage of two nested loops, two different indexes by loops and the usage of the external current indexes in the internal loops.

```

1 \pspicture*(-6,-6)(6,6)
2   \degrees[240]
3   \SpecialCoor
4   \multido{\iAngleA=1+2,\iAngleB=2+2}{120}{%
5     \pspolygon*(1;\iAngleA)(9;\iAngleA)(9;\iAngleB)(1;\iAngleB)}
6   \multido{\nHueA=1.00+-0.12,\nRadiusA=2.0+1.5}{3}{%
7     \multido{\nHueB=\nHueA+0.04,\nRadiusB=\nRadiusA+0.4}{2}{%
8       \definecolor{MyColor}{hsb}{\nHueB,1,1}}
9       \pscircle*[lineWidth=0.4, linecolor=MyColor]{\nRadiusB})}
10 \endpspicture

```

⁶⁸Look at it fixedly for few seconds, and you will see the colored circles turning fast. Of course, this is an optical illusion...



And, to finish, we redraw a picture of the Italian specialist of vision Baingio Pinna.⁶⁹ Note the usage of different kinds of indexes in the same loops and the change of the `degrees` parameter between the two loops.

```

1 \def\PstMyDiamond#1#2{%
2   \psset{dimen=middle,unit=0.1,linewidth=2\pslinewidth,arrows=c-c}%
3   \psline(0,-3)(-2,#1)(0,3)
4   \psline[linecolor=white](0,-3)(2,#2)(0,3)}%
5
6 \pspicture(-5,-5)(5,5)
7   \SpecialCoor
8   \psframe*[linecolor=lightgray](-5,-5)(5,5)
9   \psdot[dotstyle=B+,dotscale=3]
10  \degrees[48]
11  \multido{\iAngle=1+2,\nRotation=4.5+2.0}{24}{%
12    \rput{\nRotation}(3.5;\iAngle){\PstMyDiamond{-1}{1}}}

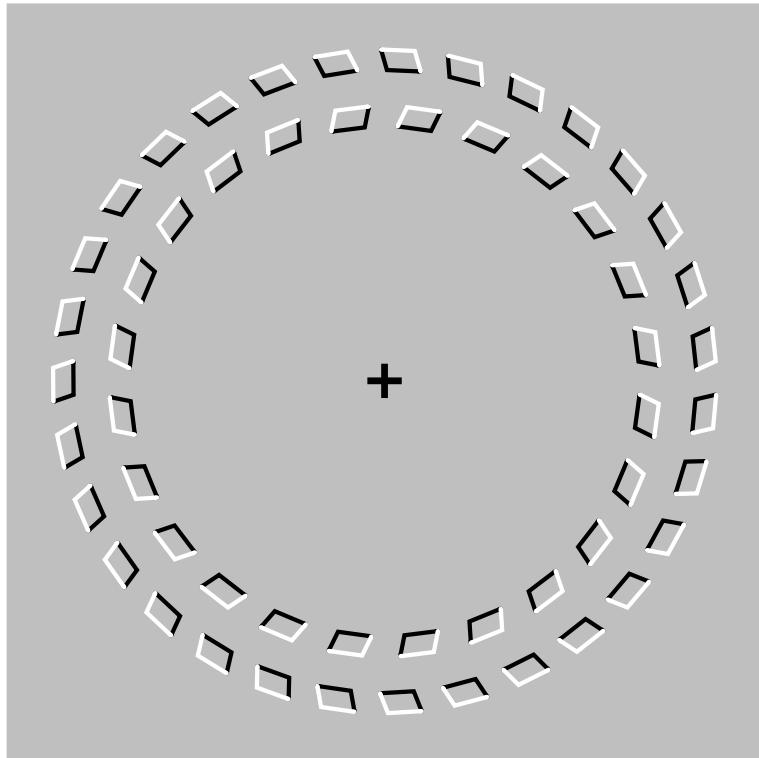
```

⁶⁹Fix the cross in the middle of the figure and put away the paper from your eyes (or move backward your head if you look at the picture on a screen): you will see the two rings of diamonds turn in opposite direction. Of course, this is too an optical illusion... You can easily find in publications, or by asking for Internet references, some explanations on this phenomena and the preceding one.

```

13 \degrees[62]
14 \multido{\iAngle=1+2,\nRotation=-3.5+2.0}{31}{%
15   \rput{\nRotation}(4.25,\iAngle){\PstMyDiamond{1}{-1}}}
16 \endpspicture

```



E Recursion

Recursion is a classical way of programming, which consist for a function to call itself with different parameters. Recursive algorithms are generally shortest than iterative ones, and are generally considered as elegant. Nevertheless, they are usually less efficient than iterative ones when executed on a computer and can easily require huge computing resources, both computing time and specially memory. So such programs must be used with care. But, for some problems, they offer a very attractive and easy way to solve complex problems.

We give here some classical examples on fractals, for the well-known Sierpinsk triangle (see for instance [49]) then for the 13 segments auto-similar curve shown by Benoit Mandelbrot in his book *The Fractal Geometry of Nature* [38].

```

1 % From the source of LaTeX 2e (latex.ltx file)
2 \long\def\@firstoftwo#1#2{#1}
3 \long\def\@secondoftwo#1#2{#2}
4

```

```

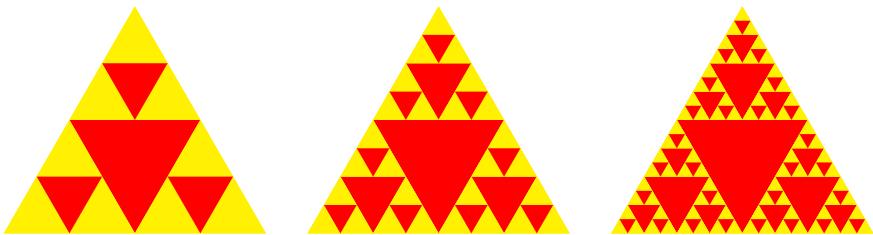
5  % The recursion macro used (from David Carlisle)
6  \def\Recursion#1{%
7    #1\relax
8    \expandafter\@firstoftwo
9  \else
10   \expandafter\@secondoftwo
11 \fi}
12
13 \def\PstFractalBegin{}% Default value empty
14
15 % General definition of a fractal
16 \def\PstFractal{\def\pst@par{}\pst@object{PstFractal}}
17
18 \def\PstFractal@i#1{%
19   \use@par% Assignment of local parameters
20   \PstFractalBegin% The first level can need special thing
21   \Recursion{\ifnum#1>\@ne}
22     {\pst@cnth=#1\relax
23      \advance\pst@cnth\m@ne
24      \PstFractalRepeat{\pst@cnth}}
25     {\PstFractalDefinition}}}
26
27 % Examples on classical fractals
28 % -----
29
30 % Sierpinski triangle
31 \def\PstSierpinskiTriangle#1{%
32   \def\PstFractalDefinition{%
33     \pspolygon*[linecolor=\PstSierpinskiExternalColor](1;0)(1;1)(1;2)
34     \rput{-2}(0,0){%
35       \pspolygon*[linecolor=\PstSierpinskiInternalColor]
36         (0.5;0.5)(0.5;1.5)(0.5;2.5)}}%
37   \def\PstFractalBegin{\PstFractalDefinition}%
38   \def\PstFractalRepeat##1{%
39     \rput(0.5;0){\PstFractal[unit=0.5]{##1}}
40     \rput(0.5;1){\PstFractal[unit=0.5]{##1}}
41     \rput(0.5;2){\PstFractal[unit=0.5]{##1}}}%
42   \pspicture(-0.866,-0.5)(0.866,1)
43     \SpecialCoor
44     \rput{90}(0,0){%
45       \degrees[3]
46       \PstFractal{#1}}
47   \endpspicture}
48
49 \def\PstMandelbrotCurve#1#2{%
50   \def\PstFractalDefinition{%
51     \pscustom[#2]{%
52       \code{1 setlinejoin}%
53       \moveto(0,0)
54       \lineto(0.5,0.9)\lineto(1,1.8)\lineto(2,1.8)\lineto(2.5,0.9)
55       \lineto(2,1.2)\lineto(1.5,1.5)\lineto(1,1.2)\lineto(1,0.6)
56       \lineto(2,0.6)\lineto(1.5,0.3)\lineto(1,0)\lineto(2,0)\lineto(3,0)}}%
57   %

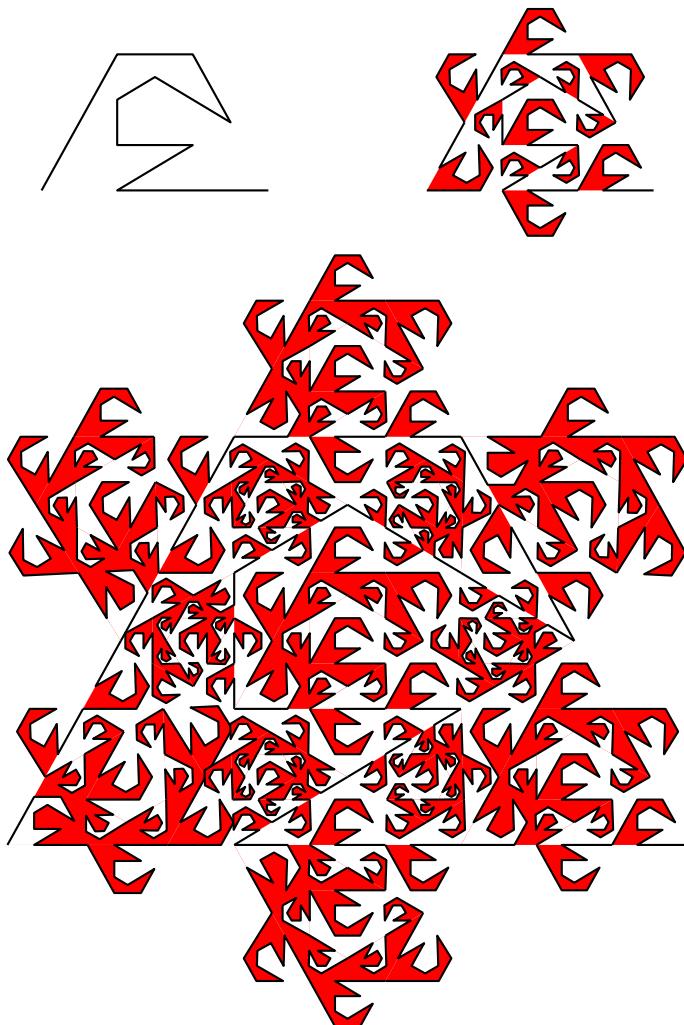
```

```

58 \def\PstFractalRepeat##1{%
59 \psset{fillstyle=solid}%
60 \rput{60.95}(0,0){%
61   \scalebox{1 -1}{\PstFractal[unit=0.3432,fillcolor=white]{##1}}}
62 \rput{60.95}(0.5,0.9){\PstFractal[unit=0.3432,fillcolor=red]{##1}}
63 \rput{0}(1,1.8){\PstFractal[unit=0.3333,fillcolor=red]{##1}}
64 \rput{-60.95}(2,1.8){\PstFractal[unit=0.3432,fillcolor=red]{##1}}
65 \rput{-210.96}(2.5,0.9){\PstFractal[unit=0.1944,fillcolor=red]{##1}}
66 \rput{-210.96}(2,1.2){%
67   \scalebox{1 -1}{\PstFractal[unit=0.1944,fillcolor=white]{##1}}}
68 \rput{-149.04}(1.5,1.5){%
69   \scalebox{1 -1}{\PstFractal[unit=0.1944,fillcolor=white]{##1}}}
70 \rput{-90}(1,1.2){%
71   \scalebox{1 -1}{\PstFractal[unit=0.2,fillcolor=white]{##1}}}
72 \rput(1,0.6){\PstFractal[unit=0.3333,fillcolor=red]{##1}}
73 \rput{-149.04}(2,0.6){\PstFractal[unit=0.1944,fillcolor=red]{##1}}
74 \rput{-149.04}(1.5,0.3){%
75   \scalebox{1 -1}{\PstFractal[unit=0.1944,fillcolor=white]{##1}}}
76 \rput(1,0){%
77   \scalebox{1 -1}{\PstFractal[unit=0.3333,fillcolor=red]{##1}}}
78 \rput(2,0){\PstFractal[unit=0.3333,fillcolor=red]{##1}}}}%
79 %
80 \PstFractal{#1}%
81
82 \bgroup
83 \def\PstSierpinskiInternalColor{red}%
84 \def\PstSierpinskiExternalColor{yellow}%
85 \psset{unit=2}%
86 \PstSierpinskiTriangle{2}\hfill%
87 \PstSierpinskiTriangle{3}\hfill%
88 \PstSierpinskiTriangle{4}%
89 \egroup
90
91 \PstMandelbrotCurve{1}%
92 \hspace{2cm}
93 \PstMandelbrotCurve{2}{fillstyle=solid,fillcolor=red}%
94
95 \bgroup
96 \psset{unit=3}%
97 \PstMandelbrotCurve{2}{linestyle=none,fillstyle=solid,fillcolor=red}%
98 \PstMandelbrotCurve{3}%
99 \egroup

```





F Computations

Computations are permanently required for computational graphics. Of course, as seen in many examples of this manual, \TeX has some capabilities to make computations on integer and real values (for reals, using *dimensions*), but there are rather huge restrictions. With PSTricks, complicated computations can be done in PostScript, which has a lot more capabilities and operands than \TeX in this area, but with the restriction that the results of these computations cannot be used in the \TeX code, as they will be known only during the interpretation of the PostScript code.

Sometimes, it is really needed to make the computations in the \TeX code. PSTricks extend a little the \TeX features in this area, adding the `\pst@divide` macro to do divisions between two dimensions (that is to say real numbers) and the `\pst@getsinandcos` macro (defined in the ‘`pst-3d`’ package) to return the sinus and cosinus of an angle. If there is more to do, external packages must be used, as the ‘`fptools`’ [18] or ‘`realcalc`’ [73] packages.

Nevertheless, in this area, the ‘fp’ package [19] from Michael Mehlich, that we will use in the next example, is from far the more powerful available.⁷⁰

We give here an example on the generation of basic auto-similar fractals, which will used the general \Recursion macro defined in the preceding section. The code is probably not very straightforward to understand, as it is a program which generate a program, but it illustrate the great capabilities of a general purpose programming language for computational graphics. In fact, it works in two steps: first, the program read the definition of the *pattern* of a fractal, which is the list of the segments⁷¹ which represent the fractal at its first iteration. Then it generate the code which define the fractal, in two parts, one which define the drawing of the fractal itself and the other which define the recursive code to apply. Then the initial program execute this generated program.

For the example of the famous Van Koch curve (see for instance [49]), the initial pattern is $(0,0)(1,0)(1.5,0.866)(2,0)(3,0)$, and we must also give for input data the scaling factor for this fractal, which is here $\frac{1}{3}$. Then the generated code for the definition of the fractal will be simply:⁷²

```

1 \pscustom{%
2   \moveto(0,0)
3   \lineto(1,0)
4   \lineto(1.5,0.866)
5   \lineto(2,0)
6   \lineto(3,0)
7 }
```

and the generated code for the recursion will be:

```

1 \rput{0}(0,0){\PstFractal[unit=0.33333]{#1}}
2 \rput{59.99927}(1,0){\PstFractal[unit=0.33333]{#1}}
3 \rput{-59.99927}(1.5,0.866){\PstFractal[unit=0.33333]{#1}}
4 \rput{0}(2,0){\PstFractal[unit=0.33333]{#1}}
```

A more sophisticated way to proceed would have been to do not store the generated codes inside macros but to write them in two files. This would

⁷⁰Take care that it has an incompatibility with the ‘multido’ package – see the paragraph M.3 in this chapter for a patch which solve this problem.

⁷¹This version of the code does not support discontinuous segments, as in the *Cantor set*, which was historically the probable first fractal discovered, by the German mathematician Georg Cantor in 1883 (see for instance [49]).

⁷²This is specially important in such circumstances to use low level macros like the \moveto and \lineto here, because fractals are complicated graphics which handle many segments, then can generate very huge files for a significant deep recursion level. For the next examples, the output file is twice huge if the \psline macro is used!

have the advantage to be able to reuse the generated codes for fractals already handled. This is not really more complicated to program, but the code is significantly longer, so we will not do it here.

And as the computing of the rotation angle for all the repeated patterns require some trigonometric computations, we will use the ‘fp’ package [19] for that (see the macro \PstFractal@GenerateCommands@iii).

```

1  \def\@enamedef#1{\expandafter\edef\csname #1\endcsname}
2
3  % To compute hypotenuse
4  \def\FPhypotenuse#1#2#3{%
5    \FPmul{\FPquada}{#2}{#2}
6    \FPmul{\FPquadb}{#3}{#3}
7    \FPadd{\FPsumquad}{\FPquada}{\FPquadb}
8    \FProot{\FP@tmp}{\FPsumquad}{2}
9    \global\let\NetFP\FP@tmp\FP@tmp%
10   \let#1\FP@tmp
11
12  % General definition of a fractal
13  \def\PstFractal{\def\pst@par{}\pst@object{PstFractal}}
14
15  \def\PstFractal@i#1{%
16    \use@par% Assignment of local parameters
17    \Recursion
18    {\ifnum#1>@\ne}
19    {\pst@cnth=#1\relax
20     \advance\pst@cnth\m@ne
21     \multido{\i=@\ne+@\ne}{20}{% No more than 20 segments expected
22      \nameuse{PstFractal@Repeat\i}{\the\pst@cnth}}}
23    {\pscustom{%
24      \multido{\i=@\ne+@\ne}{20}{\nameuse{PstFractal@Definition\i}}}}}
25
26  % Initialisation of internal macros for the fractal definition
27  \Multido{\i=1+1}{20}{%
28    \namedef{PstFractal@Definition\i}{}%
29    \namedef{PstFractal@Repeat\i}{}}
30
31  % Macro for automatic generation of auto-similar fractals
32  % by definition of the pattern only
33  \def\PstFractal@SelfSimilar{%
34    \def\pst@par{}%
35    \pst@object{PstFractal@SelfSimilar}}
36
37  \def\PstFractal@SelfSimilar@i#1#2{%
38    \use@par% Affectation of local parameters
39    \def\PstFractal@Deep{#1}%
40    \def\PstFractal@Scale{#2}%
41    \PstFractal@GenerateCommands}
42
43  \pst@cnta=\z@% Counter for the number of segments
44

```

```

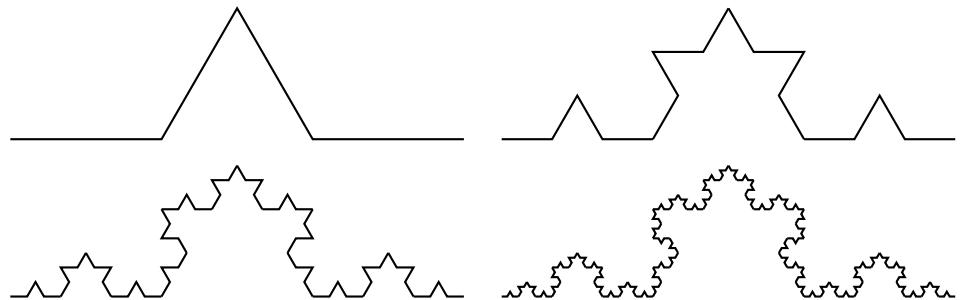
45 \def\PstFractal@GenerateCommands(#1){%
46 \advance\pst@cnta\@ne
47 \PstFractal@GenerateCommands@ii(#1)
48 \@ifnextchar({\PstFractal@GenerateCommands}%
49 {\PstFractal{\PstFractal@Deep}})}
50
51 \def\PstFractal@GenerateCommands@ii(#1,#2){%
52 \ifnum\pst@cnta>\@ne
53 \PstFractal@GenerateCommands@iii
54 (\PstFractal@PreviousX,\PstFractal@PreviousY,#1,#2)
55 \fi
56 \def\PstFractal@PreviousX{#1}%
57 \def\PstFractal@PreviousY{#2}%
58
59 \def\PstFractal@GenerateCommands@iii(#1,#2,#3,#4){%
60 \FPeval{\Xdiff}{abs(#3 - #1)}
61 \FPeval{\Ydiff}{abs(#4 - #2)}
62 \FPhypotenuse{\PstFractalHypotenuse}{\Xdiff}{\Ydiff}
63 % \FPeval easily generate overflows on next computations,
64 % so we rather use here the \FPupn variant
65 \FPupn{\PstFractalRotation}
66 {\PstFractalHypotenuse} \Xdiff / arccos 57.29578 *}
67 \ifdim#1p@<#3p@
68 \ifdim#2p@>#4p@
69 \FPneg{\PstFractalRotation}\PstFractalRotation}
70 \fi
71 \else
72 \ifdim#2p@<#4p@
73 \FPadd{\PstFractalRotation}{\PstFractalRotation}{180}
74 \else
75 \FPSub{\PstFractalRotation}{180}\PstFractalRotation}
76 \fi
77 \FPneg{\PstFractalRotation}\PstFractalRotation}
78 \fi
79 \FPround{\PstFractalRotation}{\PstFractalRotation}{5}
80 % Using \moveto and \lineto macros rather than \psline will optimize
81 % resulting file with a noticeable factor (around 2) !
82 \ifnum\pst@cnta=\tw@
83 @enamedef{\PstFractal@Definition\the\pst@cnta}{%
84 \noexpand\moveto(#1,#2)
85 \noexpand\lineto(#3,#4)}%
86 \else
87 @enamedef{\PstFractal@Definition\the\pst@cnta}{%
88 \noexpand\lineto(#1,#2)
89 \noexpand\lineto(#3,#4)}%
90 \fi
91 @enamedef{\PstFractal@Repeat\the\pst@cnta}##1{%
92 \noexpand\rput{\PstFractalRotation}{#1,#2}{%
93 \noexpand\PstFractal[unit=\PstFractal@Scale]{##1}}}
94
95 \def\PstVonKochCurve{%
96 \@ifnextchar[{\PstVonKochCurve@i}{\PstVonKochCurve@i[]}}}
97

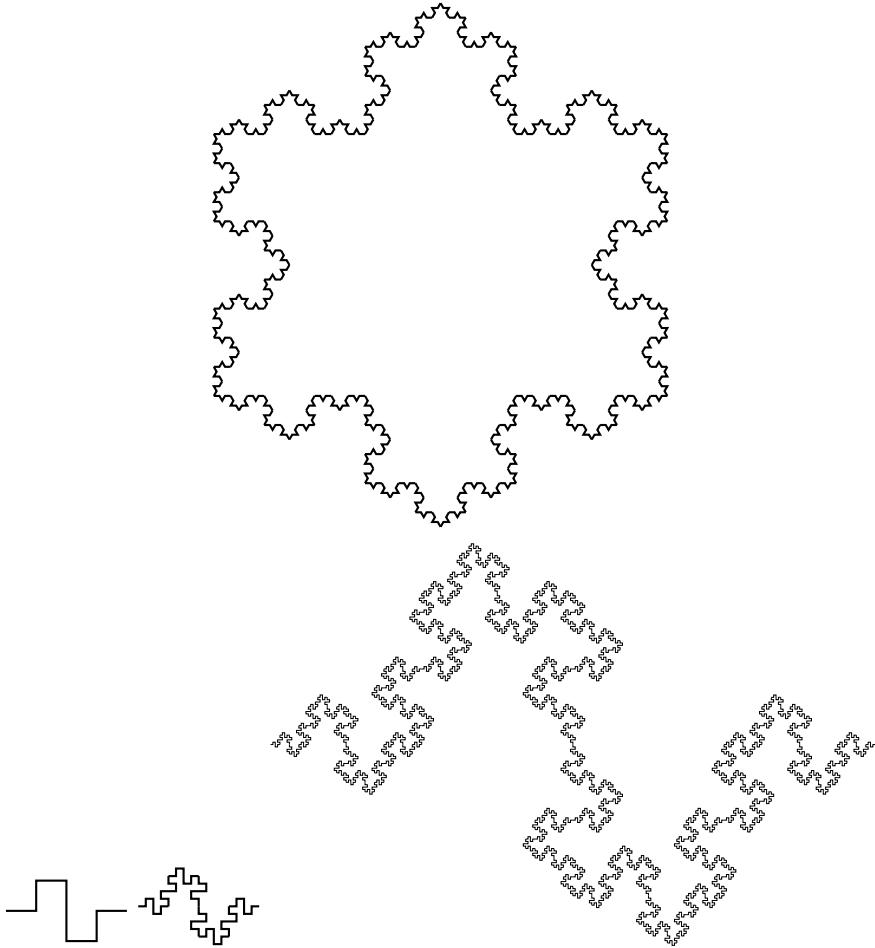
```

```

98 \def\PstVonKochCurve@i[#1]#2{%
99  \PstFractal@SelfSimilar[#1]{#2}{0.33333}{(0,0)(1,0)(1.5,0.866)(2,0)(3,0)}
100
101 \def\PstVonKochSnake{%
102  \@ifnextchar[{\PstVonKochSnake@i}{\PstVonKochSnake@i[]}}
103
104 \def\PstVonKochSnake@i[#1]#2{%
105  \rput(0,0){\PstVonKochCurve[#1]{#2}}
106  \rput{240}(3,0){\PstVonKochCurve[#1]{#2}}
107  \scalebox{-1 1}{\rput{240}{\PstVonKochCurve[#1]{#2}}}}
108
109 \def\PstMinkowskiCurve{%
110  \@ifnextchar[{\PstMinkowskiCurve@i}{\PstMinkowskiCurve@i[]}}
111
112 \def\PstMinkowskiCurve@i[#1]#2{%
113  \PstFractal@SelfSimilar[#1]{#2}{0.25}%
114  (0,0)(1,0)(1,1)(2,1)(2,0)(2,-1)(3,-1)(3,0)(4,0)}
115
116 \psset{unit=2}%
117
118 \pspicture(3,0.9)\PstVonKochCurve{1}\endpspicture
119 \hfill
120 \pspicture(3,0.9)\PstVonKochCurve{2}\endpspicture
121
122 \pspicture(3,0.9)\PstVonKochCurve{3}\endpspicture
123 \hfill
124 \pspicture(3,0.9)\PstVonKochCurve{4}\endpspicture
125
126 \pspicture(0,-2.6)(3,0.9)\PstVonKochSnake{4}\endpspicture
127
128 \pspicture(0,-0.2)(0.8,0.2)\PstMinkowskiCurve[unit=0.2]{1}\endpspicture
129 \hfill
130 \pspicture(0,-0.23)(0.8,0.23)\PstMinkowskiCurve[unit=0.2]{2}\endpspicture
131 \hfill
132 \pspicture(0,-1.3)(4,1.3)
133  \PstMinkowskiCurve[linewidth=0.2\pslinewidth]{4}
134 \endpspicture

```





G Storing and retrieving data in arrays

For some applications, it is very convenient to manage data in *arrays*, in the meaning of the classical programming languages. For that, the ‘arrayjob’ package [4] from Zuhuan Jiang is very well adapted and powerful.⁷³ It allows to store and retrieve data in mono or bi-dimensional arrays. Here is an example which allows to build easily a generic macro to draw simple (without exploded slices, etc.) two dimensional pie charts, just defining three arrays of data for the values, labels and styles of the slices.⁷⁴

```

1 \def\PstPieChart#1#2#3#4{%
2 % #1 = Values, #2 = Labels, #3 = Styles, #4 = Starting angle
3 \pspicture(-2,-2)(2,2)

```

⁷³Take care that it has an incompatibility with the ‘array’ L^AT_EX package, with a name clash on the `\array` macro that both define... You could find a workaround for this problem in the `pst-user.cls` file of this documentation.

⁷⁴The ‘pst-chrt’ contribution package [57], to draw various kinds of 2d and pseudo-3d business charts, implement such techniques.

```

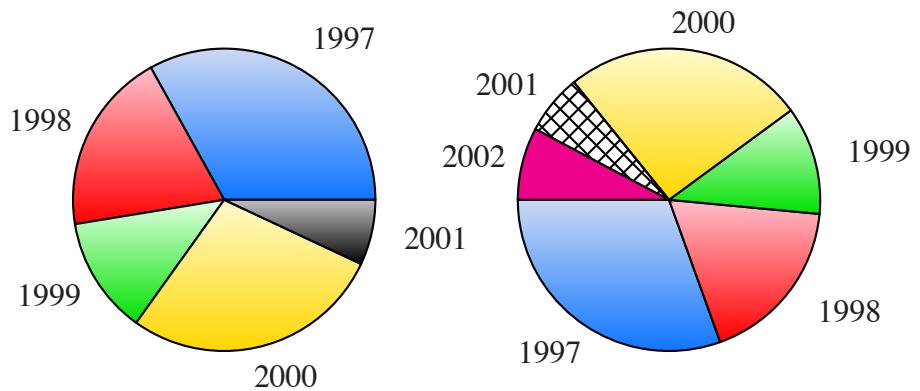
4   % To compute in \pst@dimd the sum of all the values
5   \pst@dimd=\z@
6   \Multido{\iNbValues=\z@+@\ne}{9999}{%
7     \csname check#1\endcsname(\multidocount)%
8     \ifemptydata
9       \multidostop
10    \else
11      \advance\pst@dimd by \cachedata\p@
12    \fi}
13   \degrees[\pst@number{\pst@dimd}]
14   % To compute the starting angle
15   \pst@dimh=\pst@dimd
16   \divide\pst@dimh by 360
17   \multiply\pst@dimh by #4
18   % Loop on slices
19   \multido{}{\iNbValues}{%
20     % Slice
21     \pst@dimg=\pst@dimh
22     \csname check#1\endcsname(\multidocount)%
23     \advance\pst@dimh by \cachedata\p@
24     \csname check#3\endcsname(\multidocount)%
25     \pswedge[style=\cachedata]{2}
26       {\pst@number{\pst@dimg}}{\pst@number{\pst@dimh}}
27     % Label
28     \advance\pst@dimg\pst@dimh
29     \divide\pst@dimg\tw@
30     \put{2.2}{[\pst@number{\pst@dimg}](0,0){%
31       \large\csname #2\endcsname(\multidocount)}}
32   \endpspicture}
33
34 \newarray{Values}
35 \newarray{Labels}
36 \newarray{Styles}
37
38 \readarray{Values}{32.7 & 19.3 & 12.4 & 27.6 & 6.9}
39   % Don't leave blanks between labels and between styles!
40 \readarray{Labels}{1997&1998&1999&2000&2001}
41 \readarray{Styles}{StyleGradientA&StyleGradientB&%
42   StyleGradientC&StyleGradientD&StyleGradientE}
43
44 \newpsstyle{}{}% For undefined styles
45 \newpsstyle{StyleGradientA}{%
46   fillstyle=gradient,gradbegin=LightBlue,gradend=NavyBlue}
47 \newpsstyle{StyleGradientB}{%
48   fillstyle=gradient,gradbegin=Pink,gradend=red}
49 \newpsstyle{StyleGradientC}{%
50   fillstyle=gradient,gradbegin=PaleGreen,gradend=ForestGreen}
51 \newpsstyle{StyleGradientD}{%
52   fillstyle=gradient,gradbegin=LemonChiffon,gradend=Gold}
53 \newpsstyle{StyleGradientE}{%
54   fillstyle=gradient,gradbegin=lightgray,gradend=black}
55 \newpsstyle{StyleHatchA}{fillstyle=crosshatch}
56 \newpsstyle{StyleSolidA}{fillstyle=solid,fillcolor=magenta}

```

```

57 \psset{gradmidpoint=1}%
58 \PstPieChart{Values}{Labels}{Styles}{0}
59 \hspace{1.5cm}
60 \Styles(5)={StyleHatchA}
61 \Values(6)={8.3}\Labels(6)={2002}\Styles(6)={StyleSolidA}
62 \PstPieChart{Values}{Labels}{Styles}{180}
63
64

```



Then another example to draw a classical pyramid of ages in a generic way, automatically computing the size of the graphic and the axes.

```

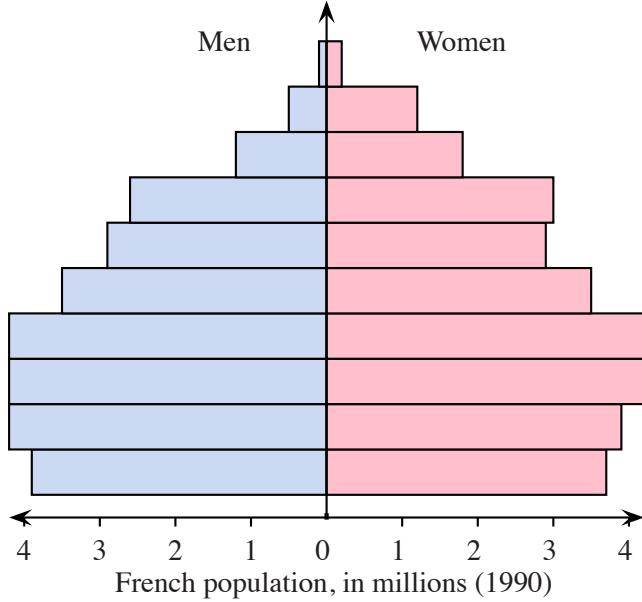
1 \def\PstPyramidAges#1#2#3#4{%
2 % #1 = Values for first set, #2 = Values for second set,
3 % #3 = Labels, #4 = Styles
4 \psset{dimen=middle}%
5 % To store in \pst@tempc the maximum value for the first set
6 % and in \pst@tempd the maximum value for the second set.
7 % This will allow to compute the size of the graphic
8 % and the correct lengths of the axes.
9 \pst@dimg=\z@
10 \pst@dimh=\z@
11 \Multido{\iNbValues=\z@+@\ne}{9999}{%
12   \csname check#1\endcsname(\multidocount)%
13   \ifemptydata
14     \multidostop
15   \else
16     \ifdim\pst@dimg<\cachedata\p@
17       \pst@dimg=\cachedata\p@
18     \fi
19     \csname check#2\endcsname(\multidocount)%
20     \ifdim\pst@dimh<\cachedata\p@
21       \pst@dimh=\cachedata\p@
22     \fi
23   \fi}
24 \edef\pst@tempc{\pst@number{\pst@dimg}}%
25 \edef\pst@tempd{\pst@number{\pst@dimh}}%
26 \pspicture(-\pst@tempc,-2.5)(\pst@tempd,\iNbValues.9)
27 % The two styles used

```

```

28 \csname check#4\endcsname{@ne}%
29 \let\PstStyleFirstSet\cachedata
30 \csname check#4\endcsname{tw@}%
31 \let\PstStyleSecondSet\cachedata
32 % Loop on values
33 \multido{\iValue=\z@+@\ne}{\NbValues}{%
34   % First set
35   \csname check#1\endcsname{\multidocount}%
36   \psframe[style=\PstStyleFirstSet]
37   (0,\iValue)(-\cachedata,\multidocount)
38   % Second set
39   \csname check#2\endcsname{\multidocount}%
40   \psframe[style=\PstStyleSecondSet]
41   (0,\iValue)(\cachedata,\multidocount)}
42 % Axes
43 \def\pshlabel##1{%
44 \ifnum##1<\z@
45   \pst@cnth=##1
46   \pst@cnth=-\pst@cnth
47   \the\pst@cnth
48 \else
49   ##1
50 \fi}%
51 \psaxes[tickstyle=bottom,ticks=x,labels=x,arrowscale=2]
52 {<->}(0,-0.5)(-\pst@tempc,-0.5)(\pst@tempd,\NbValues.9)
53 % Labels and legend
54 \rput[r]{-1,\NbValues}{\csname #3\endcsname{@ne}}
55 \rput[l]{1,\NbValues}{\csname #3\endcsname{tw@}}
56 \rput(0,-2){\csname #3\endcsname{thr@@}}
57 \endpspicture}%
58
59 \newarray{\ValuesMen}
60 \newarray{\ValuesWomen}
61 \newarray{\Labels}
62 \newarray{\Styles}
63
64 \readarray{\ValuesMen}{%
65   3.9 & 4.2 & 4.2 & 4.2 & 3.5 & 2.9 & 2.6 & 1.2 & 0.5 & 0.1}
66 \readarray{\ValuesWomen}{%
67   3.7 & 3.9 & 4.2 & 4.2 & 3.5 & 2.9 & 3.0 & 1.8 & 1.2 & 0.2}
68
69 % Don't leave blanks between labels and between styles!
70 \readarray{\Labels}{Men&Women&%
71   French population, in millions (1990)}
72 \readarray{\Styles}{StyleSolidA&StyleSolidB}
73
74 \newpsstyle{StyleSolidA}{fillstyle=solid,fillcolor=LightBlue}
75 \newpsstyle{StyleSolidB}{fillstyle=solid,fillcolor=Pink}
76
77 \psset{yunit=0.6}%
78 \PstPyramidAges{\ValuesMen}{\ValuesWomen}{\Labels}{\Styles}

```



H Random

Sometimes, it is needed or useful to be able to generate random values. There are two ways to do it, at the \TeX level or at the PostScript one. In the first case, the ‘random’ package [72] from Donald Arseneau must be used, which allow to generate integer or real numbers (as *numbers* or \TeX lengths).

From the usage point of view, an important difference with the PostScript level is that the graphic will look the same until we recompile the \TeX file, at the opposite to the PostScript generation of the random numbers, which will generate a different graphic each time the file will be viewed or printed.⁷⁵ Another difference is the speed. As \TeX is very slow for computations, it could take a long time to compile the graphic if you must generate a huge amount of random numbers. Nevertheless, as explained, these numbers will not be generated again during each interpretation of the PostScript code.

We give here examples of both usage, first to generate a patchwork of little squares of a random level of gray color. In the \TeX version, we cannot use the `\multido` macro, as we need to generate at each iteration not only the code of a pure PStricks graphic object, but also the code defining the new color. In the PostScript version, we must redefine the internal `\pst@usecolor` macro.

```

1 \multido{\nXA=0.0+0.5,\nXB=0.5+0.5}{10}{%
2   \multido{\nYA=0.0+0.5,\nYB=0.5+0.5}{10}{%

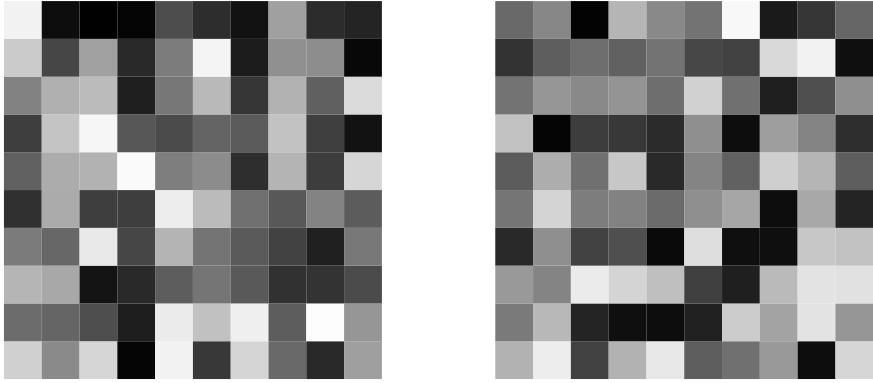
```

⁷⁵You can verify it calling two times a visualization tool on this file: the first next graphic will stay the same when the second one will be different.

```

3   \setrandim{\pst@dimh}{0pt}{1pt}%
4   \definecolor{MyGray}{gray}{\pst@number{\pst@dimh}}%
5   \psframe*[linecolor=MyGray](nXA,nYA)(nXB,nYB)}}
6   \hfill
7   \multips(0,0.5){10}{%
8     \multips(0.5,0){10}{%
9       \def\pst@usecolor{\rand 101 mod 100 div setgray }%
10      \psframe*(0.5,0.5)}}

```



In the next example, we draw colored bubbles in a test tube. The position of the bubbles is randomly computed, as their sizes, which may also depend of the value of the current unit, and they also grow according to the vertical position inside the tube. The color of the bubbles is also set randomly, using the Brightness component of the HSB color model.

```

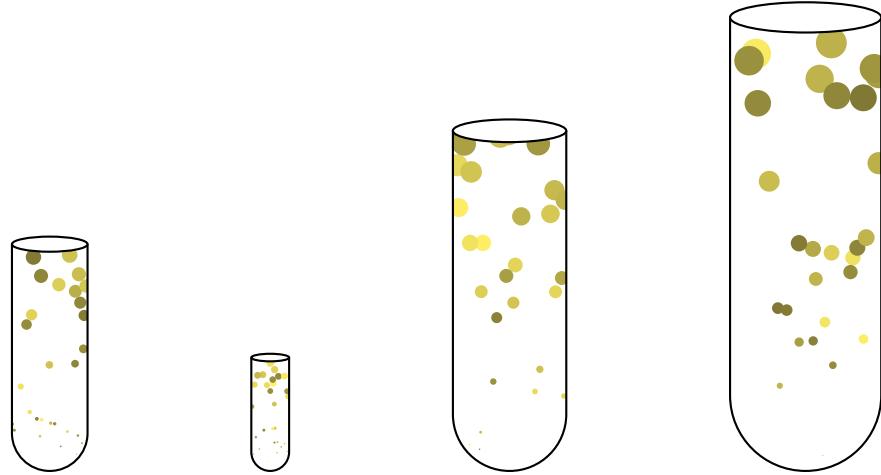
1 \def\PstTestTube{%
2   \psline[linearc=0.5]{-0.5,3}{-0.5,0}(0.5,0)(0.5,3)
3   \psellipse[fillstyle=solid,fillcolor=white]{(0,3)(0.5,0.1)}}
4
5 \def\PstTestTubeBubbles{%
6   \pspicture(-0.5,0)(0.5,3.1)
7   \psclip{\pscustom*[linestyle=none]{\PstTestTube}}
8     \multido{}{30}{%
9       \setrandim{\pst@dimc}{-0.5pt}{0.5pt}%
10      \setrandim{\pst@dimd}{0pt}{3pt}%
11      \setrandim{\pst@dimg}{0.5pt}{1pt}%
12      \definecolor{MyColor}{hsb}{0.15,0.6,\pointless{\pst@dimg}}%
13      % The size increase proportionally to the vertical position
14      \pst@dimh=\pst@dimd%
15      % And depend of the unit
16      \pst@dimh=0.02\psunit
17      \pst@dimh=\pointless{\pst@dimg}\pst@dimh
18      \psdot[dotscale=\pointless{\pst@dimh},linecolor=MyColor]
19        (! \pst@number{\pst@dimc} \pst@number{\pst@dimd})}}
20 \endpsclip
21 \PstTestTube
22 \endpspicture}
23

```

```

24 \PstTestTubeBubbles\hfill
25 {\psset{unit=0.5}\PstTestTubeBubbles}\hfill
26 {\psset{unit=1.5}\PstTestTubeBubbles}\hfill
27 {\psset{unit=2}\PstTestTubeBubbles}

```



Now, a macro to draw fuzzy lines, to mimic hand made bars.⁷⁶ It allow to draw sticks to show quantities.

```

1 \def\PstLineFuzzy[#1](#2,#3)(#4,#5)(#6,#7)(#8,#9){%
2 \pscurve[#1](! #2 rand 101 mod 1000 div sub
3 #3 rand 101 mod 1000 div sub)
4 (! #4 rand 101 mod 1000 div sub
5 #5 rand 101 mod 1000 div sub)
6 (! #6 rand 101 mod 1000 div sub
7 #7 rand 101 mod 1000 div sub)
8 (! #8 rand 101 mod 1000 div sub
9 #9 rand 101 mod 1000 div sub)}
10
11 \def\PstSticks#1{%
12 \multido{\iStick=0+1,\nXA=0.1+0.1,\nXB=-0.5+0.1,
13 \nXC=-0.35+0.10,\nXD=-0.15+0.10}{#1}{%
14 \pst@cnta=\iStick
15 \pst@cnth=\iStick
16 \divide\pst@cnth by 5
17 \multiply\pst@cnth by 5
18 \ifnum\iStick>0\relax
19 \ifnum\pst@cnta=\pst@cnth
20 \PstLineFuzzy[linecolor=red]{%
21 (\nXB,0.2)(\nXC,0.4)(\nXD,0.6)(\nXA,0.8)
22 \hbox to 0.2\psxunit{}}
23 \fi
24 \fi
25 \PstLineFuzzy[](\nXA,0.1)(\nXA,0.4)(\nXA,0.7)(\nXA,1)}}
26

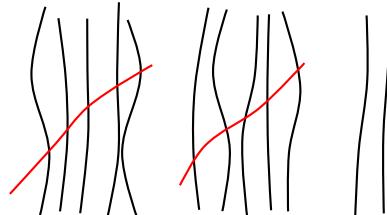
```

⁷⁶The initial idea come from MetaFun [40] from Hans Hagen.

```

27 \pspicture(-0.1,0)(3.6,3)
28   \psset{unit=3}%
29   \PstSticks{12}
30 \endpspicture
31
32 \large
33 \psset{xunit=0.8,yunit=0.3}%
34 \begin{tabular}{|l|l|p{5.5cm}|}
35   \hline
36   Linux & 27 & \PstSticks{27} \\ \hline
37   Mac OS X & 14 & \PstSticks{14} \\ \hline
38   Windows XP & 43 & \PstSticks{43} \\ \hline
39 \end{tabular}

```



Linux	27	\PstSticks{27}
Mac OS X	14	\PstSticks{14}
Windows XP	43	\PstSticks{43}

You could find another example of random usage applied on tilings in Section 50.4.

A lot of things can be done with random. To finish this section, we give an example of how to build a random mosaic,⁷⁷ of any number of divisions, from a PSTricks graphic or an external image.⁷⁸

```

1 \newarray{\PstMosaic@Indexes}
2
3 \def\PstMosaic(#1,#2)(#3,#4)#5#6{%
4   % #1,#2 = X and Y dimensions (without unit)
5   % #3,#4 = horizontal and vertical number of shifts
6   % #5 = PSTricks graphic code or external image
7   % #6 = PSTricks parameters for the borders
8   \expandarrayelementtrue
9   \dataheight=#3\relax
10  % Initialization of an array of indexes
11  \multido{\iX=\@ne+\@ne}{#3}{%

```

⁷⁷The initial idea come from MetaFun [40] from Hans Hagen.

⁷⁸Take care that in this last case the image is loaded the number of times corresponding to the number of cells, which can generate a huge resulting file if the compiler used is not able to load the file only one time, or if you do not use a special EPS file with a core part which will be loaded only one time (see the paragraph 15 (in it version 2.0) in the document of Keith Reckdahl about the insertion of EPS images with L^TE_X [75]).

```

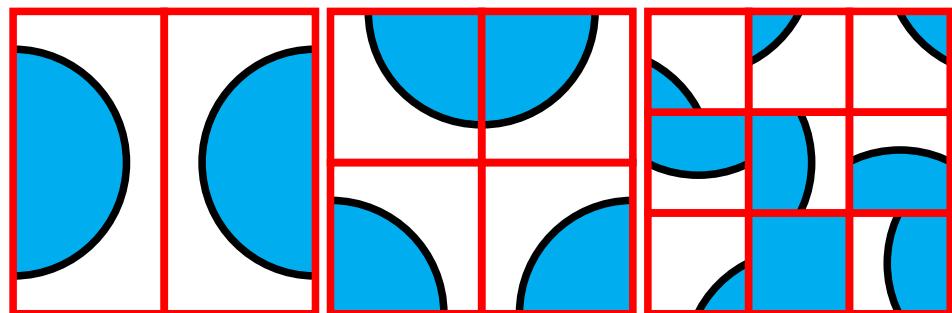
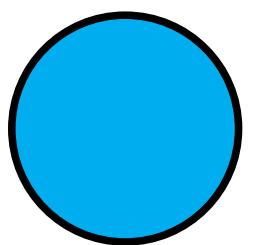
12  \multido{\iY=\@ne+\@ne}{#4}{%
13      \PstMosaic@Indexes(\iX,\iY)={(\iX,\iY)}}
14  % Shuffling of this array of indexes (50 times must be enough!)
15  \PstMosaic@IndexesShuffle(#3,#4){50}%
16  % The mosaic
17  \pst@dimg=#1\p@
18  \divide\pst@dimg by #3
19  \pst@dimh=#2\p@
20  \divide\pst@dimh by #4
21  \pst@dimtonum{\pst@dimg}\pst@tempi}%
22  \pst@dimtonum{\pst@dimh}\pst@tempj}%
23  \pspicture(#1,#2)
24      \multido{\iX=\@ne+\@ne}{#3}{%
25          \multido{\iY=\@ne+\@ne}{#4}{%
26              \checkPstMosaic@Indexes(\iX,\iY)%
27              \expandafter\PstMosaic@GetIndexes\cachedata
28              \PstMosaic@Tile(\pst@tempi,\pst@tempj)%
29                  (\pst@tempg,\pst@tempm)(\iX,\iY){#5}{#6}}}
30  \endpspicture}}
31
32 \def\PstMosaic@IndexesShuffle(#1,#2){#3}%
33 \multido{\iShuffle=\@ne+\@ne}{#3}{%
34     \setrannum{\pst@cnta}{\@ne}{#1}%
35     \setrannum{\pst@cntb}{\@ne}{#2}%
36     \setrannum{\pst@cntc}{\@ne}{#1}%
37     \setrannum{\pst@cntd}{\@ne}{#2}%
38     % Exchange of \PstMosaic@Indexes(\pst@cnta,\pst@cntb)
39     % and \PstMosaic@Indexes(\pst@cntc,\pst@cntd)
40     \checkPstMosaic@Indexes(\the\pst@cnta,\the\pst@cntb)%
41     \let\pst@tempg\cachedata
42     \checkPstMosaic@Indexes(\the\pst@cntc,\the\pst@cntd)%
43     \PstMosaic@Indexes(\the\pst@cnta,\the\pst@cntb)=\cachedata}%
44     \PstMosaic@Indexes(\the\pst@cntc,\the\pst@cntd)=\pst@tempg}}
45
46 \def\PstMosaic@GetIndexes(#1,#2){%
47 \edef\pst@tempg{#1}%
48 \edef\pst@tempm{#2}%
49
50 \def\PstMosaic@Tile(#1,#2)(#3,#4)(#5,#6){#7#8}%
51 \PstMosaic@Position{#1}{#3}\pst@tempa}%
52 \PstMosaic@Position{#2}{#4}\pst@tempb}%
53 \PstMosaic@Position{#1}{#5}\pst@tempc}%
54 \PstMosaic@Position{#2}{#6}\pst@tempd}%
55 \rput(\pst@tempc,\pst@tempd){%
56     \psclip\psframe*[linestyle=none](#1,#2)%
57         \rput(-\pst@tempa,-\pst@tempb){#7}%
58     \endpsclip
59     \psframe[dimen=middle,#8](#1,#2)}}
60
61 \def\PstMosaic@Position#1#2#3{%
62 \pst@dimh=#1\p@
63 \pst@cnth=#2
64 \advance\pst@cnth\m@ne

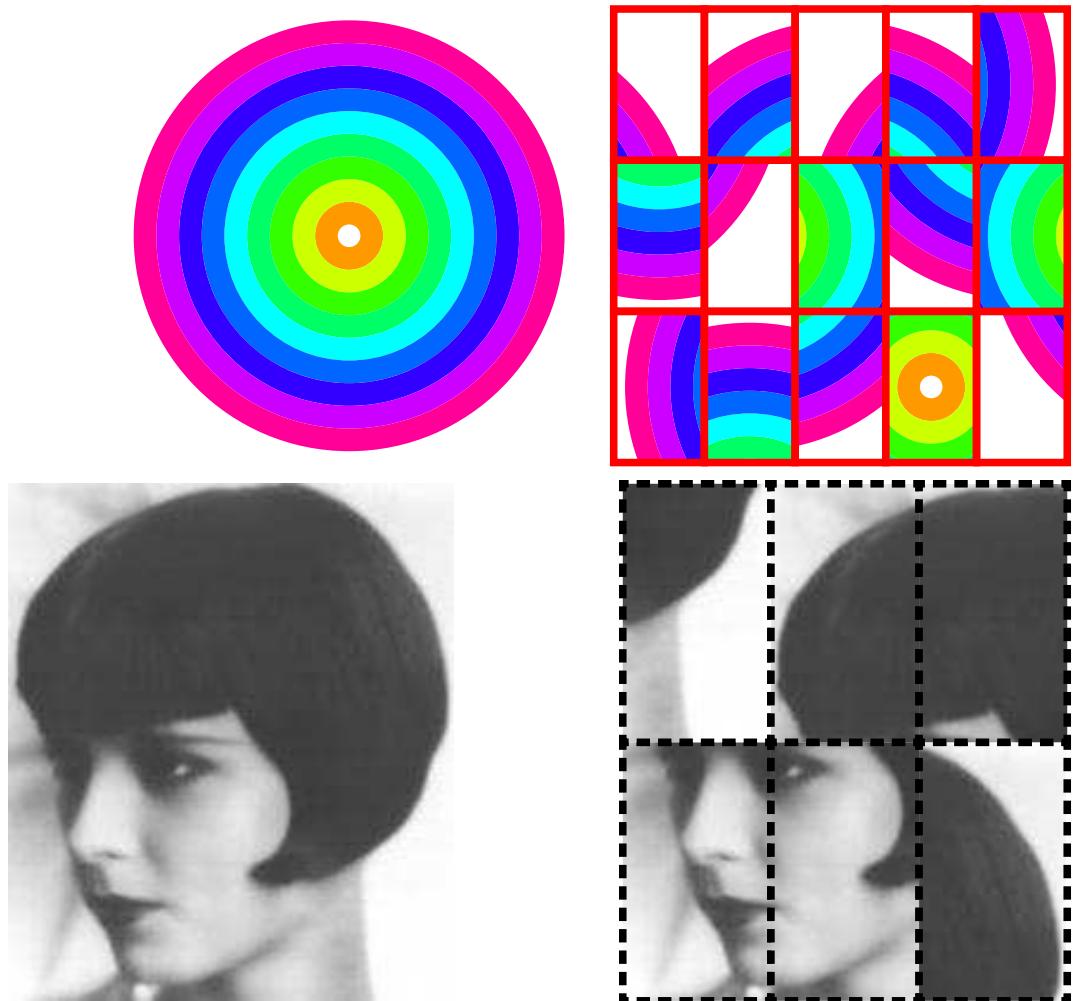
```

```

65 \multiply\pst@dimh\pst@cnth
66 \pst@dimtonum{\pst@dimh}{#3}}
67
68 % Examples
69 % -----
70
71 \def\PstCircle{%
72 \pscircle*[linewidth=0.1,fillstyle=solid,fillcolor=cyan](2,2){1.5}}
73
74 \pspicture(4,4)
75   \PstCircle
76 \endpspicture
77 \hfill
78 \PstMosaic(4,4)(2,1){\PstCircle}{linecolor=red,linewidth=0.1}
79 \hfill
80 \PstMosaic(4,4)(2,2){\PstCircle}{linecolor=red,linewidth=0.1}
81 \hfill
82 \PstMosaic(4,4)(3,3){\PstCircle}{linecolor=red,linewidth=0.1}
83
84 \def\PstCircleHue{%
85 \psset{unit=3}%
86 \multido{\nHue=0.0+0.1}{10}{%
87   \definecolor{MyColor}{hsb}{\nHue,1,1}%
88   \pscircle*[linewidth=0.1,linecolor=MyColor](1,1){\nHue}}}
89
90 \pspicture(6,6)
91   \PstCircleHue
92 \endpspicture
93 \hfill
94 \PstMosaic(6,6)(5,3){\PstCircleHue}{linecolor=red,linewidth=0.1}
95
96 \includegraphics{lbrooks}% Louise Brooks (1906-1985)
97 \hfill
98 \PstMosaic(5.88,6.84)(3,2){\includegraphics[bb=0 0 1 1]{lbrooks}}
99   {linestyle=dashed,linewidth=0.1}

```





I Coordinates

With its *special* coordinates (see Section 54), PSTricks offer powerful capabilities to handle coordinates. Nevertheless, one more thing which can be useful is to be able to handle relative coordinates, both for Cartesian and polar ones, and also for self defined lengths.

I.1 Relative coordinates

```

1 % D.G. - 1994/1997
2
3 % Minimal implementation of relative cartesian and polar
4 % coordinates
5 % (n:m) add n units in X direction and m in Y direction
6 % (n!m) add n units in X direction and m degrees of angle
7

```

```

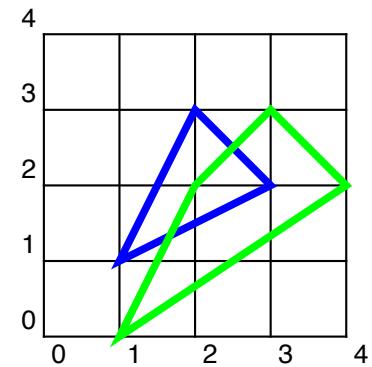
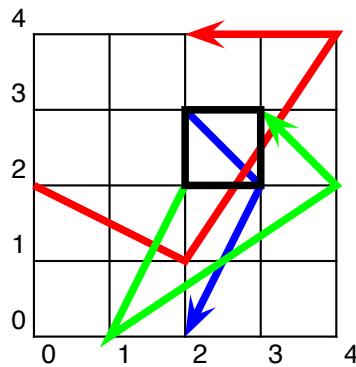
8  % New functions
9  \def\psaddtoxlength#1#2{%
10 \let\@psunit\psxunit
11 \afterassignment\pstunit@off
12 \advance#1 #2\@psunit}
13
14 \def\psaddtoylength#1#2{%
15 \let\@psunit\psyunit
16 \afterassignment\pstunit@off
17 \advance#1 #2\@psunit}
18
19 \def\psaddtoangle#1{%
20 \edef\pst@angle{\pst@angle #1 \pst@angleunit add }}
21
22 \def\PstRelativeCartesian@coor#1#2{%
23 \psaddtoxlength{\pst@dimg}{#1}%
24 \psaddtoylength{\pst@dimh}{#2}%
25 \edef\pst@coor{\pst@number\pst@dimg \pst@number\pst@dimh}}
26
27 \def\PstRelativePolar@coor#1#2{%
28 \psaddtoxlength{\pst@dimg}{#1}%
29 \psaddtoangle{#2}%
30 \edef\pst@coor{\pst@number\pst@dimg \pst@angle \tx@PtoC}}
31
32 \def\special@@@coor#1:#2:#3:@nil{%
33 \ifx#3:\relax
34   \polar@coor{#1}{#2}%
35 \else
36   \special@@@coor#1:::@nil
37 \fi}
38
39 \def\special@@@coor#1:#2:#3:@nil{%
40 \ifx#3:\relax
41   \PstRelativeCartesian@coor{#1}{#2}%
42 \else
43   \special@@@coor#1!:@nil
44 \fi}
45
46 \def\special@@@coor#1!#2:#3:@nil{%
47 \ifx#3!\relax
48   \PstRelativePolar@coor{#1}{#2}%
49 \else
50   \cartesian@coor#1,\relax,:@nil
51 \fi}

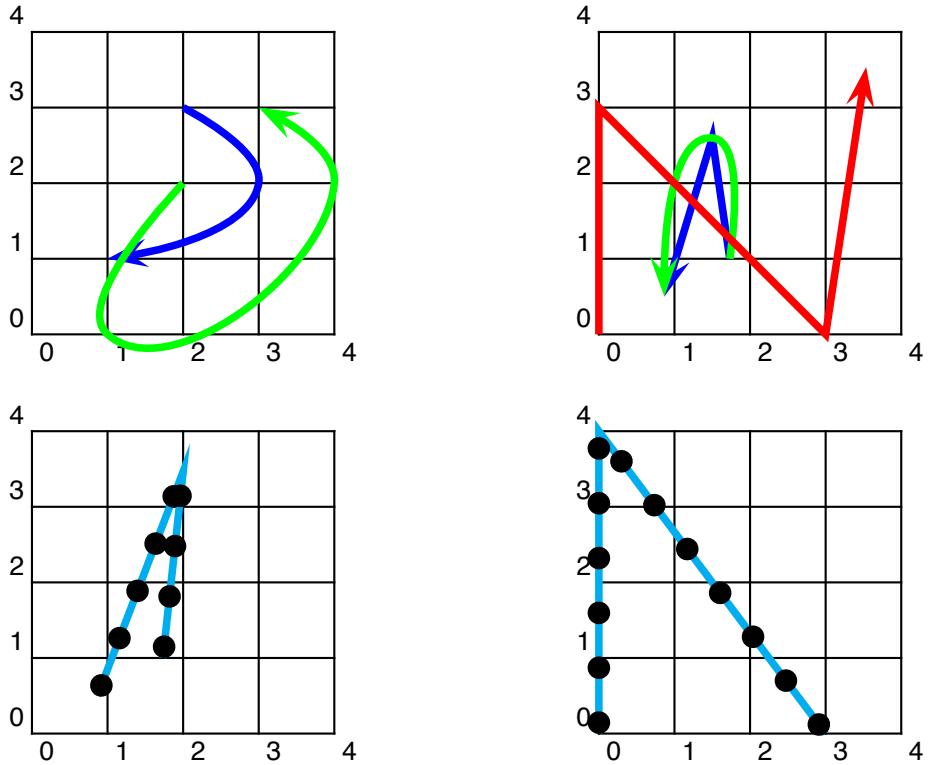
```

```

1 \psset{subgriddiv=0,linewidth=0.1,arrowscale=1.5}
2
3 \pspicture(4,4)\psgrid
4   \psline[linecolor=blue]{->}(2,3)(3,2)(2,0)
5   \psline[linecolor=red]{->}(0,2)(2:-1)(4,4)(-2:0)
6   \psline[linecolor=green]{->}(2,2)(-1:-2)(3:2)(-1:1)
7   \psframe(2,2)(1:1)
8 \endpspicture
9 \hfill
10 \pspicture(4,4)\psgrid
11   \psset{xunit=2}%
12   \pspolygon[linecolor=blue](1,3)(1.5,2)(0.5,1)
13   \pspolygon[linecolor=green](1,2)(-0.5:-2)(1.5:2)(-0.5:1)
14 \endpspicture
15
16 \pspicture(4,4)\psgrid
17   \pscurve[linecolor=blue]{->}(2,3)(3,2)(1,1)
18   \pscurve[linecolor=green]{->}(2,2)(-1:-2)(3:2)(-1:1)
19 \endpspicture
20 \hfill
21 \pspicture(4,4)\psgrid
22   \psline[linecolor=blue]{->}(2;30)(3;60)(1;30)
23   \pscurve[linecolor=green]{->}(2;30)(1!30)(-2!-30)
24   \psline[linecolor=red]{->}(0;0)(3!90)(0!-90)(2!45)
25 \endpspicture
26
27 \pspicture(4,4)\psgrid
28   \psline[linecolor=cyan](2;30)(4;60)(1;30)
29   \psline[linestyle=dotted,linewidth=0.3,dotsep=0.4](2;30)(2!30)(-3!-30)
30 \endpspicture
31 \hfill
32 \pspicture(4,4)\psgrid
33   \degrees[4]
34   \psline[linecolor=cyan](3;0)(4;1)(0;2)
35   \psline[linestyle=dotted,linewidth=0.3,dotsep=0.4](3;0)(1!1)(-4!1)
36 \endpspicture

```





Another wish could be to define relative values of existing real or lengths parameters. This can be done using a macro like the `\Pst@SetAbsOrRelDim@i` one below, that we will illustrate on a modified version of the `\PstBar` macro built in the chapter XII, which define its parameters using the interface of the ‘pst-key’ package.

```

1 \define@key{psset}{Length}{%
2   \Pst@SetAbsOrRelDim@i{\PstBar@Length}\#1\@nil}
3
4 \define@key{psset}{Height}{%
5   \Pst@SetAbsOrRelDim@i{\PstBar@Height}\#1\@nil}
6
7 \def\Pst@SetAbsOrRelDim@i#1{%
8   \@ifnextchar+{\Pst@SetAbsOrRelDim@ii{#1}}%
9     {\Pst@SetAbsOrRelDim@iii{#1}}}
10
11 \def\Pst@SetAbsOrRelDim@ii#1+#2\@nil{%
12   \pssetlength{\pst@dimh}\#1\%
13   \psaddtolength{\pst@dimh}\#2\%
14   \edef#1{\the\pst@dimh}}
15
16 \def\Pst@SetAbsOrRelDim@iii#1#2\@nil{\edef#1{\#2}}
17
18 \setkeys{psset}{Length=3,Height=1}\% Default values
19
20 \def\PstBar{\@ifnextchar[{\PstBar@i}{\PstBar@i[]})}

```

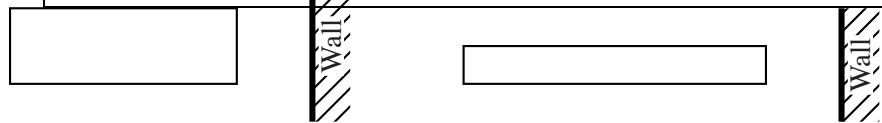
```

21 \def\PstBar@i[#1]{%
22   \setkeys{psset}{framesep=1pt,#1}%
23   \psframe(\PstBar@Length,\PstBar@Height)%
24   \pssetlength{\pst@dima}{\PstBar@Length}%
25   \psaddtolength{\pst@dima}{\@ne}%
26   \pssetlength{\pst@dimc}{\PstBar@Height}%
27   \psaddtolength{\pst@dimc}{0.5}%
28   \psline[linewidth=0.08](\pst@dima,-0.5)(\pst@dima,\pst@dimc)%
29   \pst@dimb=\pst@dima
30   \psaddtolength{\pst@dimb}{0.5}%
31   \psframe[linestyle=none,fillstyle=hlines]%
32     (\pst@dima,-0.5)(\pst@dimb,\pst@dimc)%
33   \psaddtolength{\pst@dimc}{0.25}%
34   \pssetlength{\pst@dimc}{\PstBar@Height}%
35   \divide\pst@dimc\tw@
36   \rput*{90}{\pst@dima,\pst@dimc}{Wall}}%
37 
```

```

1 \rput(-6,0){\PstBar}
2 \PstBar[Length=4,Height=0.5]

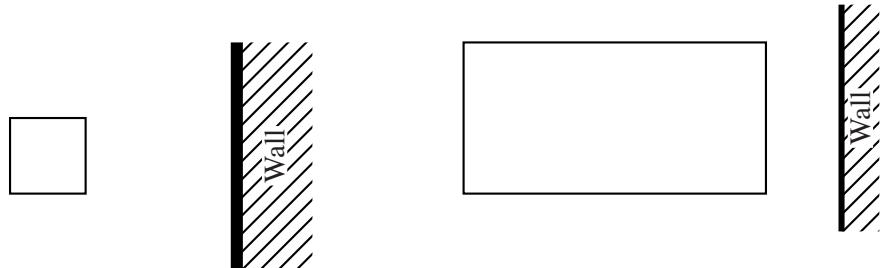
```



```

1 \rput(-6,0){\PstBar[unit=2,Length=1cm,Height=1cm]}
2 \PstBar[Length=+1cm,Height=+1cm]

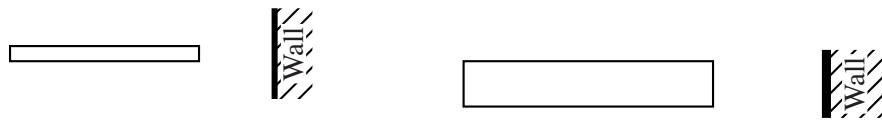
```



```

1 \rput(-6,0){%
2   \setkeys{psset}{Length=4}%
3   \PstBar[Length=-15mm,Height=-0.8cm]%
4   \PstBar[unit=1.5,Length=-0.8,Height=-1.4]

```



J Nodes

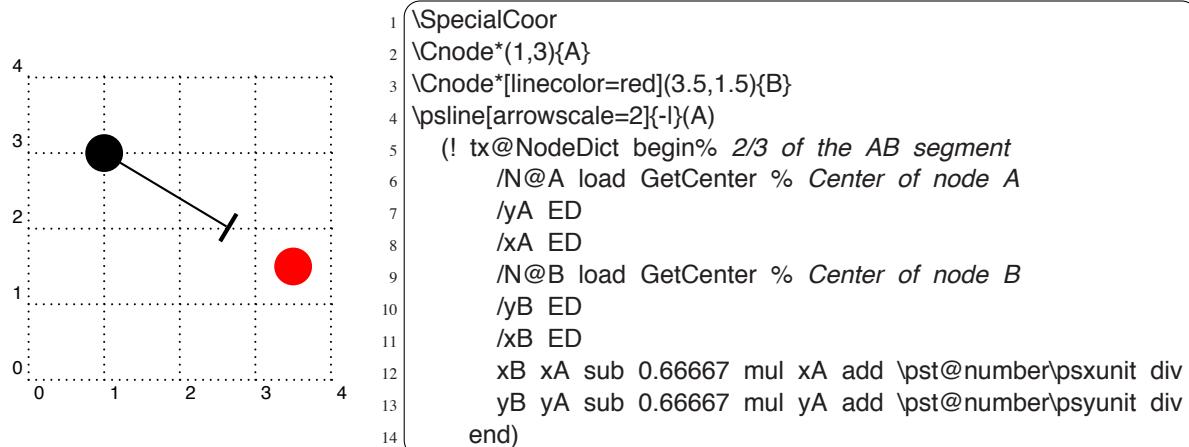
Nodes are a very powerful way to handle coordinates.⁷⁹

Nevertheless, outside their usage as *special* coordinates (see Section 54), it is also interesting to know how to retrieve the coordinates of a node and how to position objects or new nodes relatively to existing ones.

J.1 Getting the coordinates of a node

When nodes are defined, their explicit coordinates cannot be retrieved at the \TeX level, if they are unknown.⁸⁰ But they can be retrieved at the PostScript level, requiring to be handled only in this language part, which is often enough for practical needs. Here is how to proceed, using the `GetCenter` PostScript macro and the `tx@NodeDict` PostScript dictionary.

We use this function here to draw a line from a node to the two thirds of the virtual line joining this node to another one (the solution shown for that in the next paragraph is clearly easier and does not require any PostScript programming, but this is a general way to proceed, for other needs too). The PostScript sequence `/N@NodeName load GetCenter` allow to retrieve the coordinates of the node specified, leaving them on the stack. So, the `ED` (*Exchange and Define* macro), introduced in the `pstricks.pro` header file, allow to store these values in PostScript variables, for later reuse.



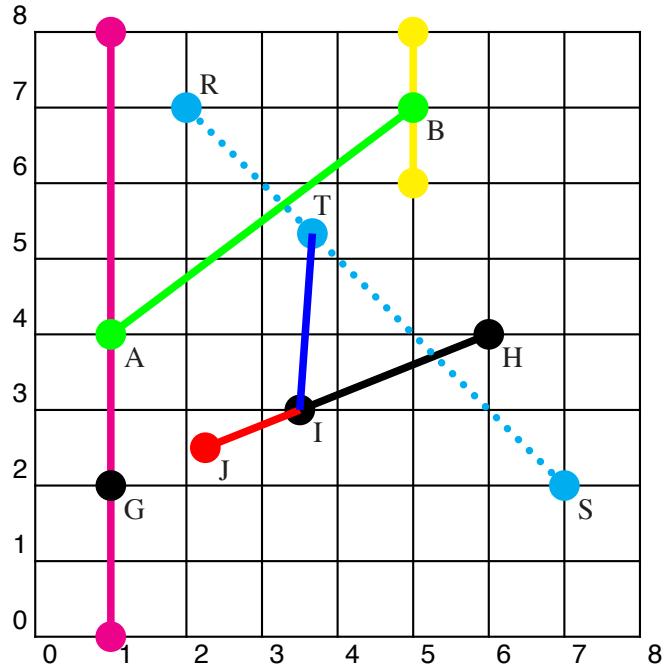
⁷⁹The contribution packages ‘`pst-circ`’ [58] from Christophe Jorssen and Herbert Voss, for electric circuits, and ‘`pst-eucl`’ [59] from Dominique Rodriguez, for Euclidean geometry drawings, are good illustrations of which high level generic functions they allow to built.

⁸⁰This is because there is no possibility to sent back informations from the PostScript level to the \TeX one during the compilation (the $\text{V}\text{\TeX}$ compiler [82] has such capability, but using such feature will give a non portable code).

J.2 Positions relative to nodes

Positioning objects or new nodes relatively to existing ones is sometimes needed, and is easy to do, with the conjunction of the `\pnode` and `\ncput` macros, together with the `tpos` parameter.

```
1 \SpecialCoor
2 \psset{linewidth=0.1,radius=0.5}%
3 \pcline[linecolor=magenta](1,0)(1,8)
4 \ncput{\pnode{A}}% A is the middle of (1,0) and (1,8) -> (1,4)
5 \pcline[linecolor=yellow](5,6)(5,8)
6 \ncput{\pnode{B}}% B is the middle of (5,6) and (5,8) -> (5,7)
7 \psline[linecolor=green](A)(B)
8 %
9 \pnode(1,2){G}
10 \pnode(6,4){H}
11 \ncline[linestyle=none]{G}{H}
12 % Join the middle of the segment joining G and H, with H
13 \ncput{\psline(H)}
14 %
15 % I is the middle of the segment joining G and H
16 \ncput{\pnode{I}}
17 % J is the middle of the segment joining G and I
18 \ncput[npos=0.25]{\pnode{J}}
19 \psline[linecolor=red](I)(J)
20 %
21 \psset{linecolor=cyan}%
22 \pnode(2,7){R}
23 \pnode(7,2){S}
24 \ncline[linestyle=dotted]{R}{S}
25 % T is the third of the segment joining R and S
26 \ncput[npos=0.3333]{\pnode{T}}
27 \psline[linecolor=blue](I)(T)
```



K Boxes

Many of the PSTricks macros have an argument for text that is processed in restricted horizontal mode (in *L^AT_EX* parlance, LR-mode) and then transformed in some way. This is always the macro's last argument, and it is written {<stuff>} in this *User's Guide*. Examples are the framing, rotating, scaling, positioning and node macros. They will be called "LR-box" macros, and use framing as the leading example in the discussion below.

In restricted horizontal mode, the input, consisting of regular characters and boxes, is made into one (long or short) line. There is no line-breaking, nor can there be vertical mode material such as an entire displayed equation. However, the fact that you can include another box means that this isn't really a restriction.

For one thing, alignment environments such as `\halign` or *L^AT_EX*'s `tabular` are just boxes, and thus present no problem. Picture environments and the box macros themselves are also just boxes. Actually, there isn't a single PSTricks command that cannot be put directly in the argument of an LR-box macro. However, entire paragraphs or other vertical mode material such as displayed equations need to be nested in a `\vbox` or *L^AT_EX* `\parbox` or `minipage`. *L^AT_EX* users should see the '`fancybox`' package [15] and its documentation for extensive tips and tricks for using LR-box commands.

The PSTricks LR-box macros have some features that are not found in most other LR-box macros, such as the standard *L^AT_EX* LR-box commands.

With *L^AT_EX* LR-box commands, the contents is always processed in text mode, even when the box occurs in math mode. PSTricks, on the other hand, preserves math mode, and attempts to preserve the math style as

well. TeX has four math styles: text, display, script and scriptscript. Generally, if the box macro occurs in displayed math (but not in sub- or superscript math), the contents are processed in display style, and otherwise the contents are processed in text style (even here the PSTricks macros can make mistakes, but through no fault of their own). If you don't get the right style, explicitly include a `\textstyle`, `\displaystyle`, `\scriptstyle` or `\scriptscriptstyle` command at the beginning of the box macro's argument.

In case you want your PSTricks LR-box commands to treat math in the same as your other LR-box commands, you can switch this feature on and off with the commands

```
\psmathboxtrue  
\psmathboxfalse
```

You can have commands (such as, but not restricted to, the math style commands) automatically inserted at the beginning of each LR-box using the

```
\everypsbox{commands}
```

command.⁸¹

If you would like to define an LR-box environment *name* from an LR-box command *cmd*, use

```
\pslongbox{name}{{cmd}}
```

For example, after

```
1 \pslongbox{MyFrame}{\psframebox}
```

you can write⁸²

```
1 \MyFrame <stuff>\endMyFrame
```

instead of

```
1 \psframebox{<stuff>}
```

It is up to you to be sure that *cmd* is a PSTricks LR-box command; if it isn't, nasty errors can arise.

Environments like have nice properties:

⁸¹This is a token register.

⁸²LATeX users can instead write:

`\begin{MyFrame}stuff\end{MyFrame}`

- The syntax is clearer when *stuff* is long.
- It is easier to build composite LR-box commands. For example, here is a framed minipage environment for L^AT_EX:

```

1 \pslongbox{MyFrame}{\psframebox}
2 \newenvironment{fminipage}%
3   {MyFrame\begin{minipage}}%
4   {\end{minipage}\endMyFrame}
```

- You include verbatim text and other \catcode tricks in *stuff*.

The rest of this section elaborates on the inclusion of verbatim text in LR-box environments and commands, for those who are interested. The ‘fancybox’ package also contains some nice verbatim macros and tricks, some of which are useful for LR-box commands.

The reason that you cannot normally include verbatim text in an LR-box commands argument is that T_EX reads the whole argument before processing the \catcode changes, at which point it is too late to change the category codes. If this is all Greek to you,⁸³ then just try this L^AT_EX example to see the problem:

```
1 \psframebox{\verb+\foo{bar}+}
```

The LR-box environments defined with \pslongbox do not have this problem because *stuff* is not processed as an argument. Thus, this works:

```

1 \pslongbox{MyFrame}{\psframebox}
2 \MyFrame \verb+\foo{bar}+\endMyFrame
```

\foo{bar}

The commands

\psverbboxtrue
\psverbboxfalse

switch into and out of, respectively, a special PSTricks mode that lets you include verbatim text in any LR-box command. For example:

```

1 \psverbboxtrue
2 \psframebox{\verb+\foo{bar}+}
```

⁸³Incidentally, some foreign language macros use \catcode tricks which can cause problems in LR-box macros.

```
\foo{bar}
```

However, this is not as robust. You must explicitly group color commands in *stuff*, and LR-box commands that usually ignore spaces that follow {<stuff>} might not do so when **\psverbboxtrue** is in effect.

L Frequently Asked Questions

DG: This whole sec-

tion is to recheck.

L.1 How to inactivate the PStricks macros

To suppress the PostScript code generate by PStricks macros, put the command

```
\PSTricksOff
```

at the beginning of your document. You should then be able to print or preview drafts of your document (minus the PostScript, and perhaps pretty strange looking) with any dvi driver.

L.2 Monochrome version of all or some graphics

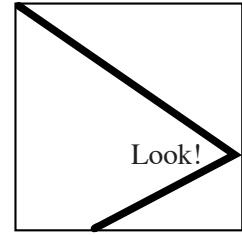
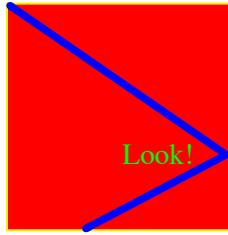
To generate a monochrome version of a paper, without having some colored graphics looking bad due to the gray conversion of the colors, it is easier to redefine the macros which set the colors rather than changing explicitly all the colors in the PStricks macros which define them. And, first, if you use PLAIN T_EX or L_AT_EX, use the **monochrome** parameter of the ‘pstcol’ package (it will just pass this option to the ‘color’ one).

```
1 \documentclass[...]{...}
2
3 \usepackage[monochrome]{pstcol}
4 %
5
6 \makeatletter
7 \def\PSTricksMonochrome{%
8 \def\psset@bordercolor##1{\pst@getcolor{white}\{\psbordercolor\}}%
9 \def\psset@doublecolor##1{\pst@getcolor{white}\{\psdoublecolor\}}%
10 \def\psset@shadowcolor##1{\pst@getcolor{darkgray}\{\psshadowcolor\}}%
11 \def\psset@linecolor##1{\pst@getcolor{black}\{\pslinecolor\}}%
12 \def\psset@fillcolor##1{\pst@getcolor{white}\{\psfillcolor\}}%
13 \def\psset@gridcolor##1{\pst@getcolor{black}\{\psgridcolor\}}%
14 \def\psset@gridlabelcolor##1{\pst@getcolor{black}\{\psgridlabelcolor\}}%
15 \def\psset@hatchcolor##1{\pst@getcolor{black}\{\pshatchcolor\}}%
16 \def\psset@subgridcolor##1{\pst@getcolor{gray}\{\pssubgridcolor\}}%
17 \makeatother
18 %
19 %\PSTricksMonochrome% To force it on the whole document
```

```

20 \begin{document}
21
22
23 \begin{pspicture}(3,3)
24   \psframe[linecolor=yellow,fillstyle=solid,fillcolor=red](3,3)
25   \psline[linecolor=blue,linewidth=0.1]{cc-cc}(0,3)(2.9,1)(1,0)
26   \rput(2,1){\textcolor{green}{Look!}}
27 \end{pspicture}
28 \hfill
29 \begin{pspicture}(3,3)
30   \PSTricksMonochrome% To force locally monochrome rendering
31   \psframe[linecolor=yellow,fillstyle=solid,fillcolor=red](3,3)
32   \psline[linecolor=blue,linewidth=0.1]{cc-cc}(0,3)(2.9,1)(1,0)
33   \rput(2,1){\textcolor{green}{Look!}}
34 \end{pspicture}
35
36 \end{document}

```



L.3 How to improve the rendering of halftones?

This can be an important consideration when you have a halftone in the background and text on top. You can try putting

```

1 \pstverb{106 45 {dup mul exch dup mul add 1.0 exch sub}
2 setscreen}

```

before the halftone, or in a header (as in headers and footers, not as in PostScript header files), if you want it to have an effect on every page.

`setscreen` is a device-dependent operator.

L.4 Including PostScript code

When you define TeX macros for including PostScript fragments in various places, all TeX macros are expanded before being passed on to PostScript. It is not always clear what this means. For example, suppose you write

```

1 \SpecialCoor
2 \def\mydata{23 43}
3 \psline(!47 \mydata add)
4 \psline(!47 \mydata\ add)

```

```
5 \psline(!47 \mydata~add)
6 \psline(!47 \mydata{} add)
```

You will get a PostScript error in each of the `\psline` commands. To see what the argument is expanding to, try use \TeX 's `\edef` and `\show`. E.g.,

```
1 \def\mydata{23 43}
2 \edef\temp{47 \mydata add}
3 \show\temp
4 \edef\temp{47 \mydata\ add}
5 \show\temp
6 \edef\temp{47 \mydata~add}
7 \show\temp
8 \edef\temp{47 \mydata{} add}
9 \show\temp
```

\TeX expands the code, assigns its value to `\temp`, and then displays the value of `\temp` on your console. Hit *return* to proceed. You will find that the four samples expand, respectively, to:

```
1 47 23 43add
2 47 23 43\ add
3 47 23 43\penalty \@M \ add
4 47 23 43{} add
```

All you really wanted was a space between the `43` and `add`. The command `\space` will do the trick:

```
1 \psline(!47 \mydata\space add)
```

You can include balance braces `{ }`; these will be passed on verbatim to PostScript. However, to include an unbalanced left or right brace, you have to use, respectively,

\pslbrace
\psrbrace

Don't bother trying `\}` or `\%`.

Whenever you insert PostScript code in a PStricks argument, the dictionary on the top of the dictionary stack is `tx@Dict`, which is PStrick's main dictionary. If you want to define your own variables, you have two options:

Simplest Always include a `@` in the variable names, because PStricks never uses `@` in its variables names. You are at a risk of overflowing the `tx@Dict` dictionary, depending on your PostScript interpreter. You are also more likely to collide with someone else's definitions, if there are multiple authors contributing to the document.

Safest Create a dictionary named `TDict` for your scratch computations. Be sure to remove it from the dictionary stack at the end of any code you insert in an argument. E.g.,

```
1 TDict 10 dict def TDict begin <your code> end
```

L.5 ConTeXt color macros support

ConTeXt fully support PSTricks. Nevertheless, as with the ‘color’ package in L^AT_EX, this is logical to rather use the standard ConTeXt syntax to define and use colors. For that, a minor adaptation of the `\dodefinedcolor` macro is needed. We also add the support of the HSB color model with the ConTeXt syntax, not currently supported by this T_EX format.

```

1 % D.G. - April 2003
2
3 % To allow to define colors in Gray, RGB, CMYK and HSB models,
4 % using the ConTeXt syntax.
5
6 \def\dodefinedcolor[#1][#2]{%
7   \doifassignmentelse{#2}{\dodefinedcolor[#1][#2]\#2\relax}{}}
8
9 \def\dodefinedcolor[#1][#2]\#3\#4\relax{%
10   \edef\tempa{#3}%
11   \edef\tempb{#4}%
12   \ifx\tempa\tempb% Gray model
13     \getparameters[pstricks][s=0,#2]%
14     \expanded{\newgray{#1}{\pstricksg}}%
15   \else
16     \edef\tempb{r}%
17     \ifx\tempa\tempb% RGB model
18       \getparameters[pstricks][r=0,g=0,b=0,#2]%
19       \expanded{\newrgbcolor{#1}{\pstricksr} {\pstricksg} {\pstricksb}}%
20     \else
21       \edef\tempb{c}%
22       \ifx\tempa\tempb% CMYK model
23         \getparameters[pstricks][c=0,m=0,y=0,k=0,#2]%
24         \expanded{\newcmykcolor{#1}{\pstricksr} {\pstricksm} {\pstricksy} {\pstricksk}}%
25       \else
26         \edef\tempb{H}%
27         \ifx\tempa\tempb% HSB model
28           \getparameters[pstricks][H=0,S=0,B=0,#2]%
29           \expanded{\newhsbcolor{#1}{\pstricksH} {\pstricksS} {\pstricksB}}%
30         \fi
31       \fi
32     \fi
33   \fi
34 \fi}
35
36 % Support of the HSB model
37 % -----
```

```

38 \def\dodododefcolor#1#2#3#4[#5][#6]{% #2==set(g)value #3==set[e]x]value
39   {#1\addtocommalist{#5}\colorlist
40    \doifassignmentelse{#6}
41     {@@resetcolorparameters
42      \getparameters[??cl@@@][#6]
43      \doifelse{@@cl@@n@@cl@@g@@cl@@b}
44        {@@cl@@z@@cl@@z@@cl@@z}
45        {\doifelse{@@cl@@c@@cl@@m@@cl@@y@@cl@@k}
46          {@@cl@@z@@cl@@z@@cl@@z@@cl@@z}
47          {\doifelse{@@cl@@H@@cl@@S@@cl@@B}
48            {@@cl@@z@@cl@@z@@cl@@z}
49            % D.G. modification begin - May. 26, 2003
50            {\doifelse{@@cl@@H@@cl@@S@@cl@@B}
51              {@@cl@@z@@cl@@z@@cl@@z}
52              % D.G. modification end
53              {\doifelse{@@cl@@s@@cl@@z
54                \showmessage\!colors8{{[#6]},#5}%
55                #3{??cr#5}{\colorZpattern}}
56                #3{??cr#5}{\colorSpattern}}}
57              % D.G. modification begin - May. 26, 2003
58              {\#3{??cr#5}{\colorHpattern}}}
59              % D.G. modification end
60              {\#3{??cr#5}{\colorCpattern}}
61              {\#3{??cr#5}{\colorRpattern}}
62              \doifdefinedelse{??cr#6}
63              {\doifelse{#5}{#6}
64                {%
65                  % this way we can freeze
66                  % \definecolor[somecolor][somecolor]
67                  % and still prevent cyclic definitions
68                  \iffreezecolors#3{??cr#5}{\.getvalue{??cr#6}}\fi
69                  \iffreezecolors@\EA#3\else@\EA#2\fi
70                  {\??cr#5{\.getvalue{??cr#6}}}
71                  \showmessage\!colors3{#5}}%
72                \ifcase#4\or
73                  \unexpanded#2{#5}{\switchtocolor{#5}}% \unexpanded toegevoegd
74                \fi}
75 \def\colorHpattern{%
0H:@@cl@@H:@@cl@@S:@@cl@@B:@@cl@@A:@@cl@@t}

```

```

1 \starttext
2
3 % New colors (in ConTeXt syntax)
4 \definecolor[ForestGreen][r=0.13,g=0.55,b=0.13]
5 \definecolor[LemonChiffon][r=1,g=0.98,b=0.8]
6 \definecolor[LightBlue][r=0.8,g=0.85,b=0.95]
7 \definecolor[LightGray][s=0.92]
8 \definecolor[LightOrange][c=0,m=0.2,y=0.4,k=0]
9
10 \pspicture(-1.5,-1.5)(4.7,1.5)
11   \psset{fillstyle=solid}%
12   \pscircle[fillcolor=yellow]{1.5}

```

```

13  \definecolor[LightBlue][r=0.8,g=0.85,b=0.95]
14  \definecolor[LightOrange][c=0,m=0.2,y=0.4,k=0]
15  \pscircle[linecolor=LightBlue,fillcolor=LightOrange]
16      (3.2,0){1.5}
17 \endpspicture
18
19 \pspicture(-3,-3)(3,3)
20 \psset{unit=3}%
21 \multido{\nHue=0.01+0.01}{100}{%
22     \definecolor[MyColor][H=\nHue,S=1,B=1]
23     \pscircle[linewidth=0.01,linecolor=MyColor]{\nHue}}
24 \endpspicture
25
26 \stoptext

```

L.6 Scope of parameters changes

When parameters are *globally* changed, using the `\psset` or the `\setkeys` macros, we may want that this change apply to a series of later commands, but that the previous values will be restored at some point. For that, the thing to do is not to keep the old values in some new macros and to reset later the parameters to these values, but to use the common T_EX grouping mechanism.

```

1 ...
2 \psset{Param=99}%
3 ...
4 \bgroup
5   \psset{Param=7.5}%
6   ...
7   \bgroup
8     \psset{Param=21}%
9     ...
10    \PstXxxx[Param=-44]%
11    ...% Param=21
12  \egroup
13  ...% Param=7.5
14 \egroup
15 ...% Param=99

```

L.7 Defining parameters using macros

This is not directly possible to define parameters using macros (such behavior is also not supported by the ‘keyval’ package [35]), as in:

```

1 \def\PstMyParameters{linecolor=red,fillstyle=solid,fillcolor=yellow}
2 \psset{\PstMyParameters}
3 \pscircle{1}

```

Here is how we can define a `\PstSet` macro which allow to do that. This is also possible to change the behavior of the `\psset` macro to it, and also to allow to use things like `\psframe[Attributes](2,2)`. To be able to use it too with the ‘keyval’ and ‘pst-key’ interface (which is encouraged to be used for all new PSTricks contribution packages!), we must modify the `\KV@setkeys` macro from ‘pst-key’ and restore the PSTricks definition of the `\use@par` one.

```

1 % D.G. - 2001 / 2003
2
3 % New \PstSet macro
4 \def\PstSet#1{\PstSet@#1,\@nil\ignorespaces}
5 \def\PstSet@#1,{%
6 \expandafter\PstSet@#1,\@nil
7 \@ifnextchar\@nil{\@gobble}{\PstSet@#1}}
8 \def\PstSet@#1,{%
9 \PstSet@#1=\@nil
10 \@ifnextchar\@nil{\@gobble}{\PstSet@#1}}
11 \def\PstSet@#1=#2\@nil{\pssetMOD{#1=#2}}
12
13 % Redefinition of \psset to accept arguments in macros
14 \let\@pssetORI\@psset
15 \def\pssetMOD#1{\@pssetORI#1,\@nil\ignorespaces}
16 \let\psset\@psset
17 \let\@psset\@psset@i
18
19 % For “keyval” and “pst-key” compatibility
20 \let\KV@setkeysORIG\KV@setkeys
21
22 \def\KV@setkeys#1#2{%
23 \edef\@tempa{#1}%
24 \edef\@tempb{\psset}%
25 \ifx\@tempa\@tempb
26   \PstSet@#2,\@nil%
27 \else
28   \KV@setkeysORIG{#1}{#2}%
29 \fi}
30
31 % We must restore the original PSTricks definition of \use@par
32 \def\use@par{%
33 \ifx\pst@par\empty\else
34 \expandafter\@psset\pst@par,\@nil
35 \def\pst@par{}%
36 \fi}
37
38 % Examples with the “pst-key” interface

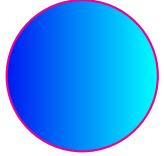
```

```

39 \define@key{psset}{Integer}{\pst@getint{\#1}{\PstInteger}}
40 \define@key{psset}{String}{\edef\PstString{\#1}}
41
42 \def\ParametersA{fillstyle=solid,fillcolor=red}
43 \def\ParametersB{linecolor=red,fillstyle=solid,fillcolor=yellow}
44 \def\ParametersC{linecolor=magenta,fillstyle=solid,fillcolor=cyan}
45 \def\ParametersD{fillstyle=gradient,gradmidpoint=1,gradangle=90}
46 \def\ParametersE{Integer=99,String=AAAA}
47 \def\ParametersF{Integer=1}
48
49 \psframebox[\ParametersA]{Monday}
\hfill
51 \psset{\ParametersB}%
52 \psframebox{Tuesday}
\hfill
54 \setkeys{psset}{\ParametersC}%
55 \psframebox{Wednesday}
\hfill
57 \psframebox[\ParametersD]{Thursday}
\hfill
59 \setkeys{psset}{\ParametersE}
60 \psframebox[\ParametersB,fillcolor=green,doubleline=true]{%
\PstString=\PstInteger}
\hfill
63 \pscircle[\ParametersD,\ParametersF]{\PstInteger}

```

Monday Tuesday Wednesday Thursday AAAA=99



L.8 Line type connections

The `setlinecap` PostScript parameter, which specify how the ends of a line finish, is accessible in PSTricks with the `c` and `C` arrows, which put this parameter to 1 and 2, respectively. But other PostScript parameters, and specially the `setlinejoin` are useful, to change the default behavior for line connections. Of course, this is still possible to use it, giving the PostScript code with the `\pscustom` macro (see Section IV).

This can be seen in such example:

```

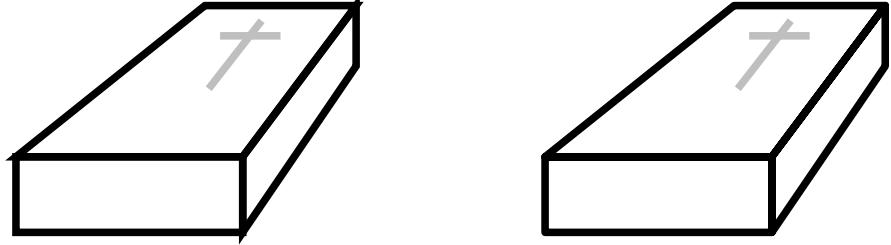
1 \psset{linewidth=0.1}
2 \pspicture(4.5,3)
3   \pspolygon(0,0)(0,1)(3,1)(3,0)
4   \pspolygon(0,1)(2.5,3)(4.5,3)(3,1)
5   \pspolygon(3,0)(3,1)(4.5,3)(4.5,2.2)
6   \psline[linecolor=lightgray](2.7,2.6)(3.5,2.6)
7   \psline[linecolor=lightgray](2.55,1.9)(3.25,2.8)
8 \endpspicture
9 \hfill
10 \pspicture(4.5,3)
11   \pscustom{%

```

```

12 \code{1 setlinejoin}
13 \pspolygon(0,0)(0,1)(3,1)(3,0)
14 \pspolygon(0,1)(2.5,3)(4.5,3)(3,1)
15 \pspolygon(3,0)(3,1)(4.5,3)(4.5,2.2)}
16 \psline[linecolor=lightgray](2.7,2.6)(3.5,2.6)
17 \psline[linecolor=lightgray](2.55,1.9)(3.25,2.8)
18 \endpspicture

```



But for intensive usage, it is easier to redefine the PStricks internal relevant macro (`\psls@solid`, for a line of the `solid` style) and to define the interesting PostScript parameters as PStricks ones (we add here the `setmiterlimit` one too).⁸⁴

```

1 % D.G. - May 2003
2
3 \def\psset@linecap#1{\pst@getint{#1}\pslinecap}
4 \psset@linecap{0}
5
6 \def\psset@linejoin#1{\pst@getint{#1}\pslinejoin}
7 \psset@linejoin{0}
8
9 \def\psset@miterlimit#1{\pst@checknum{#1}\psmiterlimit}
10 \psset@miterlimit{10}
11
12 \def\psls@solid{%
13   \pslinejoin\space setlinejoin
14   \psmiterlimit\space setmiterlimit
15   \pslinecap\space setlinecap
16   stroke }
17
18 \def\Test#1#2#3{%
19   \pspicture(3,3)\psgrid[subgriddiv=0]
20     \psset{linewidth=1}%
21     \psline[linecolor=DarkGray](0.5,1)(1.5,2)(2.5,1)
22     \psline[linecolor=LightGray, linecap=#1, linejoin=#2, miterlimit=#3]
23     (0.5,1)(1.5,2)(2.5,1)
24   \endpspicture}
25
26 \Test{0}{0}{1}\hfill\Test{0}{1}{1}\hfill\Test{0}{2}{1}
27

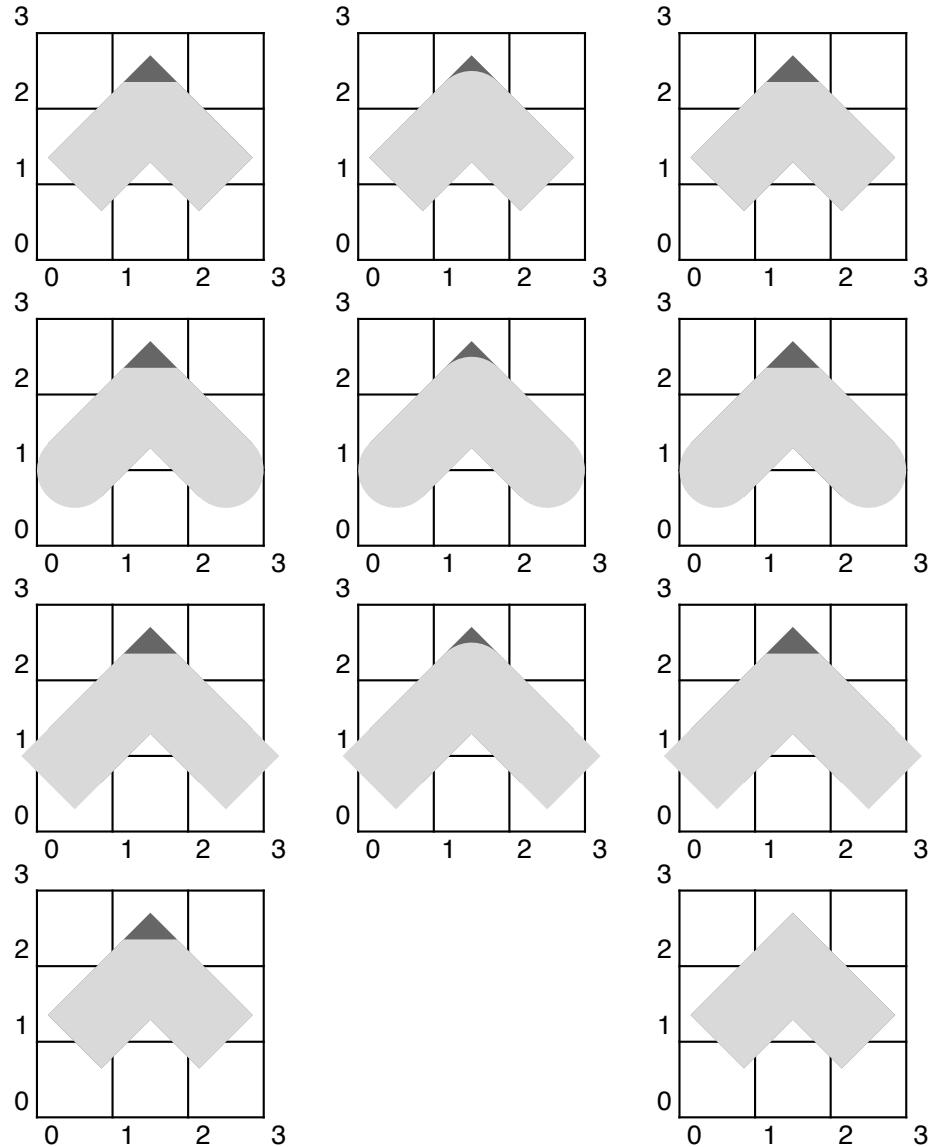
```

⁸⁴To explain the effects of these parameters, we use the same illustration than in the MetaFun manual [40].

```

28 \Test{1}{0}{1}\hfill\Test{1}{1}{1}\hfill\Test{1}{2}{1}
29
30 \Test{2}{0}{1}\hfill\Test{2}{1}{1}\hfill\Test{2}{2}{1}
31
32 \Test{0}{0}{1}\hfill\Test{0}{0}{1.5}

```



L.9 Boxes of same sizes for trees and diagrams

If we use boxes for entries in a diagram, it could be not very aesthetic to have various different sizes for them.

```

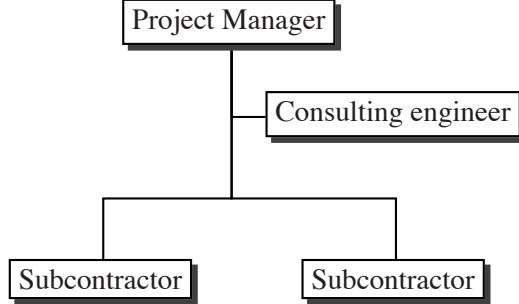
1 \def\MyNode{\psshadowbox{#1}}
2
3 \psmatrix[mnode=r,colsep=-1]
4     & [name=P]\MyNode{Project Manager} \l[-1cm]
5     & & [name=C]\MyNode{Consulting engineer} \l[0pt]

```

```

6 [name=S1]MyNode{Subcontractor} & &
7 [name=S2]MyNode{Subcontractor}
8 \endpsmatrix
9 \ncangle[angleA=-90,angleB=180]{P}{C}
10 \psset{angleA=-90,angleB=90,armB=0.8}%
11 \ncangle{P}{S1}
12 \ncangle{P}{S2}

```

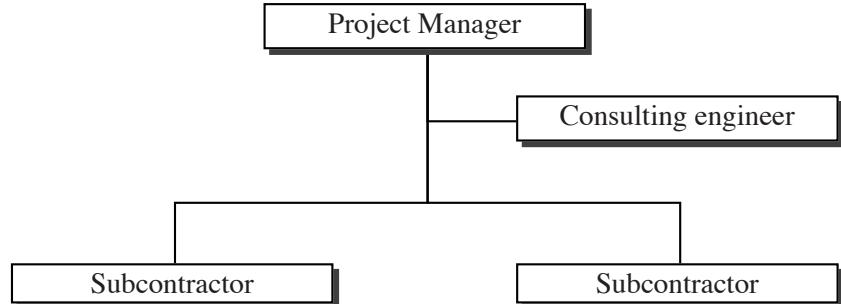


The **\makebox** L^AT_EX macro (of course, the same result can be obtain otherwise with PLAIN T_EX and ConT_EXt) allow to impose that all the boxes will have the same size.

```

1 % We force the boxes to be of the same size using \makebox
2 \def\MyNode#1{\psshadowbox{\makebox[4cm]{#1}}}
3 \psmatrix[mnode=r,colsep=-1]
4 % ...

```



Obviously, we can do the same thing in other circumstances, as here for trees (we use also a L^AT_EX syntax to compute the length of the largest label, but this can be done otherwise in PLAIN T_EX and ConT_EXt).

```

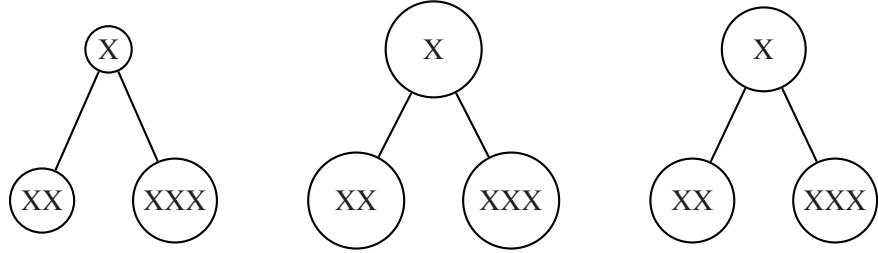
1 \def\PstMyNodeA#1{\Tcircle{#1}}
2 \def\PstMyNodeB#1{\Tcircle{\makebox[1cm]{#1}}}
3 \newlength{\PstMyLength}
4 \newcommand{\PstMyNodeC}[1]{\Tcircle{\makebox[\PstMyLength]{#1}}}
5
6 \pstree{\PstMyNodeA{X}}{\PstMyNodeA{XX}\PstMyNodeA{XXX}}
7 \hfill
8 \pstree{\PstMyNodeB{X}}{\PstMyNodeB{XX}\PstMyNodeB{XXX}}
9 \hfill

```

```

10 % Using LaTeX syntax and the 'calc' package
11 \setwidth{\PstMyLength}{XXX}%
12 \pstree{\PstMyNodeC{X}}{\PstMyNodeC{XX}\PstMyNodeC{XXX}}

```



L.10 Reference points

In structured diagrams built with the **\psmatrix** environment and trees, the default usage of the center of the nodes to draw the connections could produce non aesthetic results. In such cases, changing locally the reference point definition could produce better results.

```

1 \psmatrix[mnode=r]
2   [name=A] \psframebox{AAAA} & \\
3   & [name=B] \psframebox{BBBB}
4 \endpsmatrix
5 \ncline{A}{B}
6 \hfill
7 \psmatrix[mnode=r]
8   [name=A,ref=br] \psframebox{AAAA} \\
9   & [name=B,ref=tl] \psframebox{BBBB}
10 \endpsmatrix
11 \ncline{A}{B}

```



```

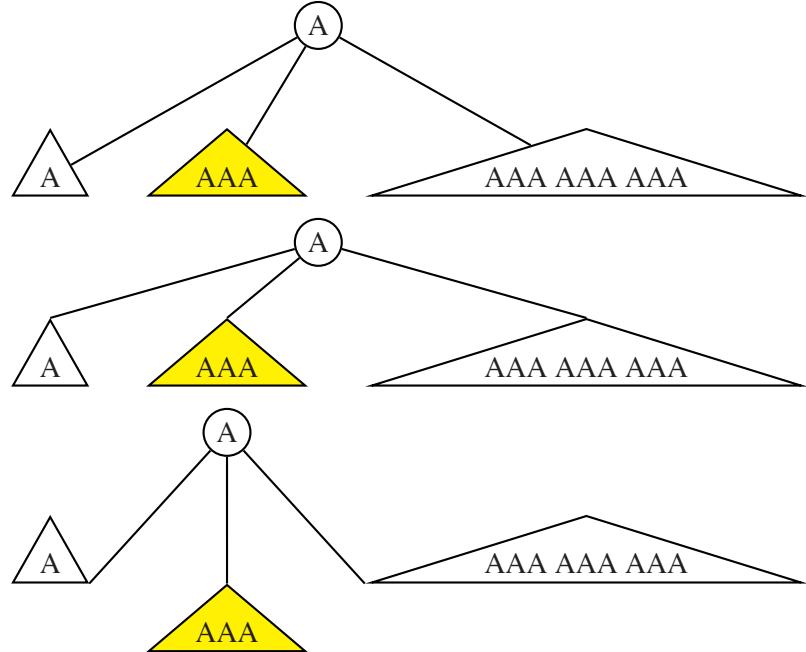
1 \pstree{\Tcircle{A}}
2   {\Ttri{A}}
3     \Ttri[fillstyle=solid,fillcolor=yellow]{AAA}
4     \Ttri{AAA AAA AAA}
5
6 \newcommand{\PstMyTtri}[2]{\Tr[ref=t]\pstribox[#1]{#2}}
7
8 \pstree[levelsep=1]{\Tcircle{A}}
9   {\PstMyTtri{A}}
10  \PstMyTtri[fillstyle=solid,fillcolor=yellow]{AAA}
11  \PstMyTtri{AAA AAA AAA}

```

```

12 \pstree{\TCircle{A}}
13   {\Tr[ref=br]{\pstriobox{A}}
14     \Tr[ref=t]{\pstriobox[fillstyle=solid,fillcolor=yellow]{AAA}}
15     \Tr[ref=bl]{\pstriobox{AAA AAA AAA}}}
16

```



L.11 Dashed and dotted hatched styles

Predefined hatched styles (see Section III) contains only continuous lines. It is easy to add dashed ones (as it is more difficult to add dotted ones, we do not include the necessary code here, which is too long for it little interest).

```

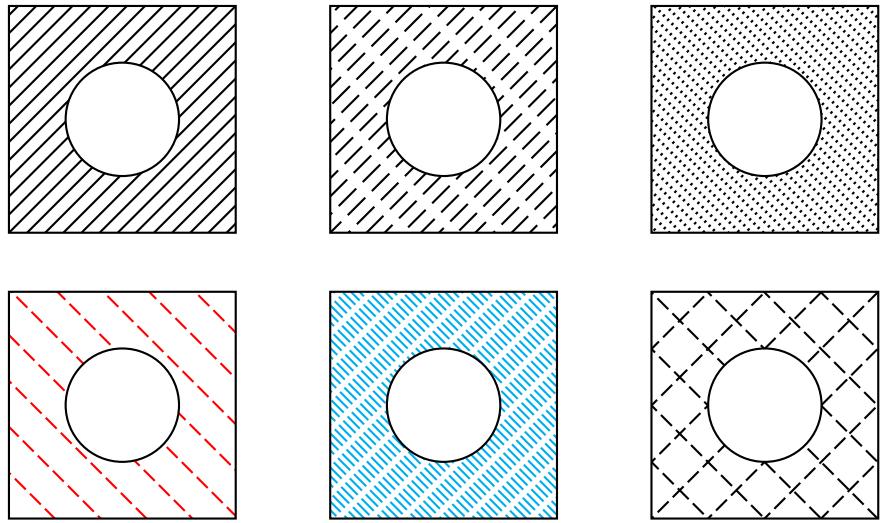
1 % D.G. - September 2002
2
3 % To define dashed and dotted hatched styles
4 \def\pshs@solid{0 setlinecap }
5 \def\pshs@dashed{[ \psk@dash ] 0 setdash }
6 \def\psset@hatchstyle#1{%
7   \@ifundefined{pshs@#1}%
8     {@pstrickserr{Hatch style '#1' not defined}\@eha}%
9   {\edef\pshatchstyle{#1}}}
10
11 \psset@hatchstyle{solid}
12
13 \def\pst@linefill{%
14   \@nameuse{pshs@\pshatchstyle}%
15   \psk@hatchangle rotate
16   \psk@hatchwidth SLW
17   \pst@usecolor\pshatchcolor
18   \psk@hatchsep

```

```

19 \tx@LineFill}
20
21 \def\ltx@TestHatchedStyle#1{%
22 \pspicture(3,3)
23   \psframe[#1](3,3)
24   \pscircle[fillstyle=solid,fillcolor=white](1.5,1.5){0.75}
25 \endpspicture}
26
27 \ltx@TestHatchedStyle{fillstyle=hlines}
28 \hfill
29 \ltx@TestHatchedStyle{fillstyle=hlines,hatchstyle=dashed,dash=0.3 0.15}
30 \hfill
31 \ltx@TestHatchedStyle{fillstyle=hlines,hatchstyle=dashed,dash=0.01 0.10,
32           hatchsep=0.05}
33
34 \ltx@TestHatchedStyle{fillstyle=vlines,hatchstyle=dashed,
35           hatchsep=0.4,hatchcolor=red}
36 \hfill
37 \ltx@TestHatchedStyle{fillstyle=vlines,hatchstyle=dashed,
38           hatchsep=0.05,dash=0.2 0.1,hatchcolor=cyan}
39 \hfill
40 \ltx@TestHatchedStyle{fillstyle=crosshatch,hatchstyle=dashed,hatchsep=0.5}

```



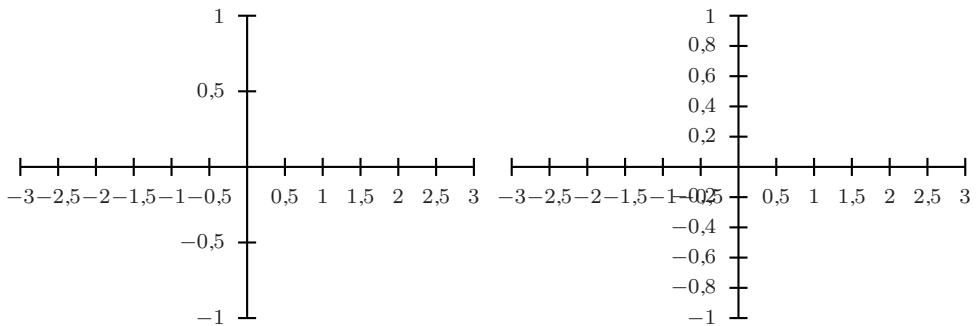
L.12 Axes with labels using commas as separators

In some languages, like French and German, this is not the point character which is used to separate the decimal part of a floating point number, but a comma. To change this convention in the **\psaxes** macro of the ‘pst-plot’ package, the **\pshlabel** and **\psvlabel** macros must be redefined, calling a utility macro which replace the decimal point by a comma (this **\PstLabelComma** macro is part of the contribution file ‘pst-plox’ which contains some extensions of the core file ‘pst-plot’).

```

1 % D.G. - June 1997
2
3 \def\PstLabelComma{\#1.\#2.\#3}@nil{%
4 \ifx\#1@empty\else$\#1$\fi
5 \ifx\#2@empty\else ifnum#2=0\else,$\#2$\fi\fi}
6
7 % To change decimal points by commas in axe labels of \psaxes
8 \def\pshlabel#1{\expandafter\PstLabelComma#1..\@nil}
9 \def\psvlabel#1{\expandafter\PstLabelComma#1..\@nil}
10
11 \scriptsize
12
13 \pspicture(-3,-2.2)(3,2.2)
14   \psaxes[Dx=0.5,Dy=0.5,yunit=2](0,0)(-3,-1)(3,1)
15 \endpspicture
16 \hfill
17 \pspicture(-3,-2.2)(3,2.2)
18   \psaxes[Dx=0.5,Dy=0.2,yunit=2](0,0)(-3,-1)(3,1)
19 \endpspicture

```



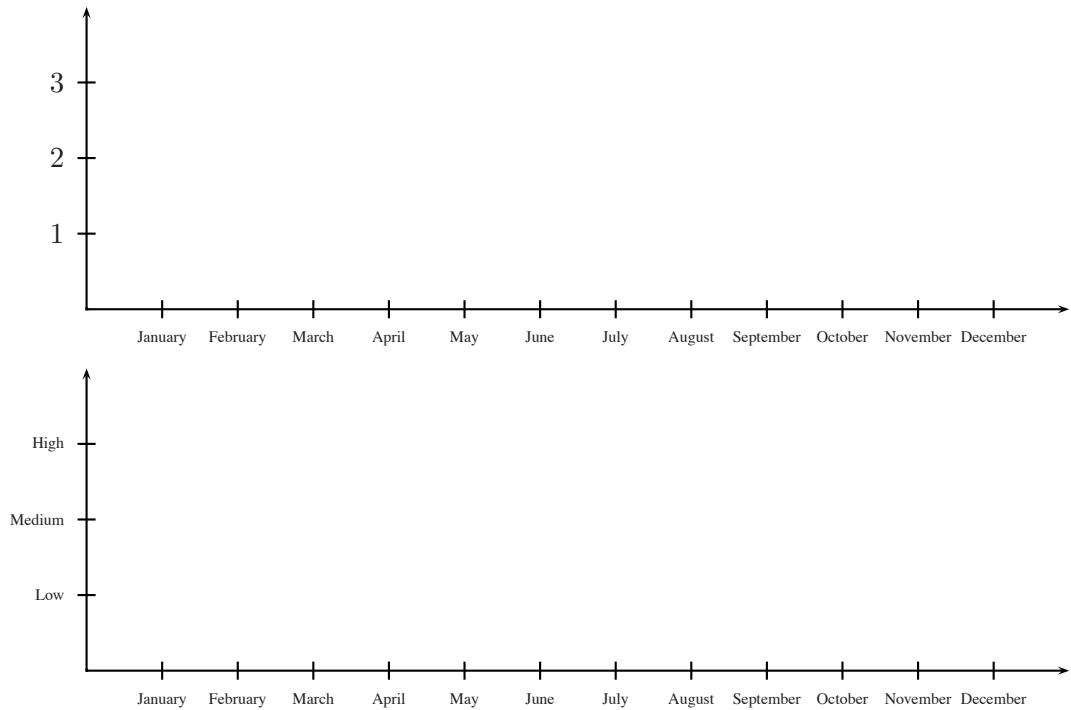
L.13 Alphabetic labels on axes

This is easy to use alphabetic labels rather than numeric ones on axes, redefining the **\pshlabel** and **\psvlabel** macros and using the ‘arrayjob’ package [4].

```

1 \newarray{Months}
2 \readarray{Months}{January&February&March&April&May&June&%
3 July&August&September&October&November&December}
4
5 \newarray{Levels}
6 \readarray{Levels}{Low&Medium&High}
7
8 \def\pshlabel#1{\tiny\Months(\#1)}
9 \psaxes[showorigin=false]{->}(13,4)
10
11 \def\psvlabel#1{\tiny\Levels(\#1)}
12 \psaxes[showorigin=false]{->}(13,4)

```



L.14 Two different axes

This is also not difficult to put two different axes on the same graph.

```

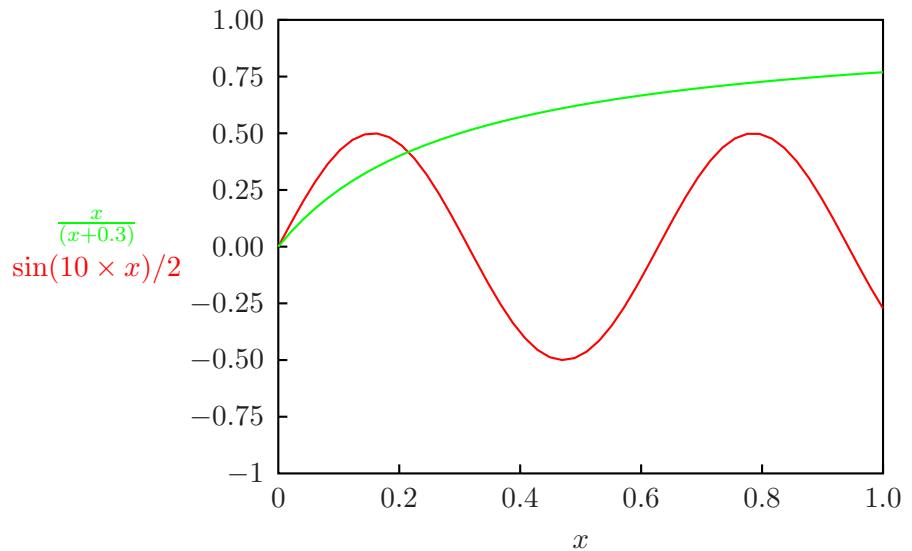
1 \psset{xunit=8,yunit=3}%
2
3 \pspicture(-0.5,-1.5)(1,1.5)
4   \psaxes[Dx=0.2,Oy=-1,Dy=0.25,tickstyle=top,
5             axesstyle=frame](0,-1)(1,1)
6   \rput(0.5,1.3){Curves example}
7   \rput(-0.3,0){%
8     \shortstack{\textcolor{green}{\frac{x}{(x + 0.3)}}\\
9       \textcolor{red}{\sin (10 \times x) / 2}}}
10  \rput(0.5,-1.3){$x$}
11  \psplot[linecolor=red]{ 1 radian = 57.296 degrees
12    {0}{1}{x 10 mul 57.296 mul sin 0.5 mul}%
13    \psplot[linecolor=green]{0}{1}{x 0.3 x add div}%
14 \endpspicture
15
16 % Same curves, but with different origin and axe for the first plot
17 \pspicture(-0.5,-1.5)(1,1.5)
18   \def\psvlabel#1{\small\textcolor{green}{#1}}%
19   \psaxes[Dx=0.2,Oy=-1,Dy=0.25,tickstyle=top,
20             axesstyle=frame](0,-1)(1,1)
21   \def\psvlabel#1{\small\textcolor{blue}{#1}}%
22   \psaxes[Oy=-1.25,Dy=0.25,tickstyle=bottom,
23             ticks=y,labels=y](1,-1)(0,1)
24   \rput(0.5,1.3){%
25     \shortstack{Curves example\\(with two different axes)}}
```

```

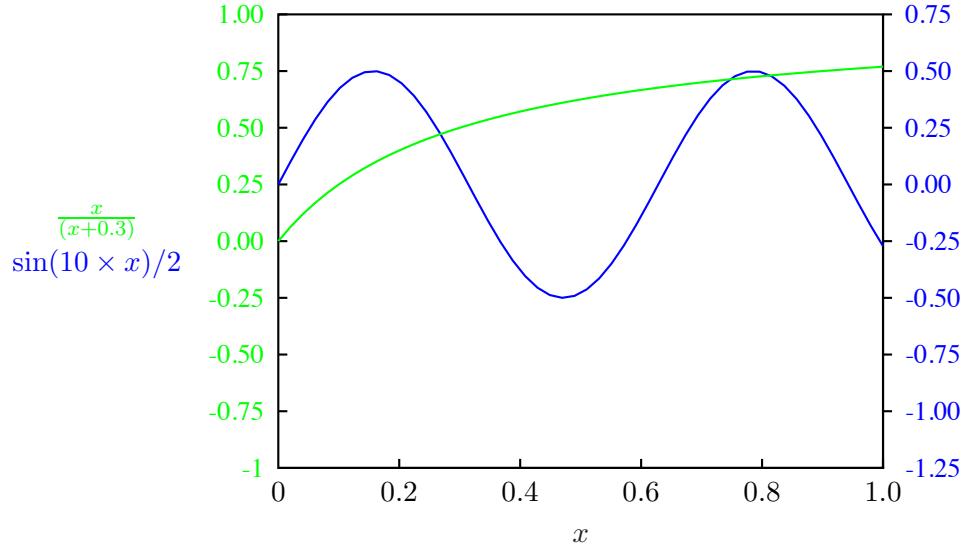
26 \rput(-0.3,0){%
27   \shortstack{\textcolor{green}{\frac{x}{(x+0.3)}}\\
28     \textcolor{blue}{\sin (10 \times x) / 2}}}
29 \rput(0.5,-1.3){$x$}
30 \rput(0,0.25){%
31   \psplot[linecolor=blue]{10 mul sin 0.5 mul}%
32   {0 1 x 10 mul 57.296 mul sin 0.5 mul}%
33   \psplot[linecolor=green]{0 1 x 0.3 x add div}%
34   {x / (x + 0.3)}
\endpspicture

```

Curves example



Curves example
(with two different axes)



DG: Add examples of logarithmic axes and different tick styles, without the too long code.

L.15 Polar Plots

Polar plots can be done with a small modification of the `\psplot` macro.⁸⁵

```

1 % D.G. - June 1998 (according the way suggested by Ulrich Dirr)
2
3 % For polar plots
4 \newif\ifpolarplot
5 \def\psset@polarplot#1{\@nameuse{polarplot#1}}
6 \psset@polarplot{false}
7
8 \def\psplot@i#1#2#3{%
9 \pst@killglue
10 \begingroup
11 \use@par
12 \{@nameuse{beginplot@\psplotstyle}%
13 \ifpolarplot
14   \addto@pscode{%
15     \psplot@init
16     /x #1 def
17     /x1 #2 def
18     /dx x1 x sub \psk@plotpoints div def
19     /xy {%
20       #3 dup x cos mul exch x sin mul
21       \pst@number\psxunit mul exch

```

⁸⁵This code was implemented in 1998 according the way suggested by Ulrich Dirr. And the examples 2, 3 and 4 come also from him.

```

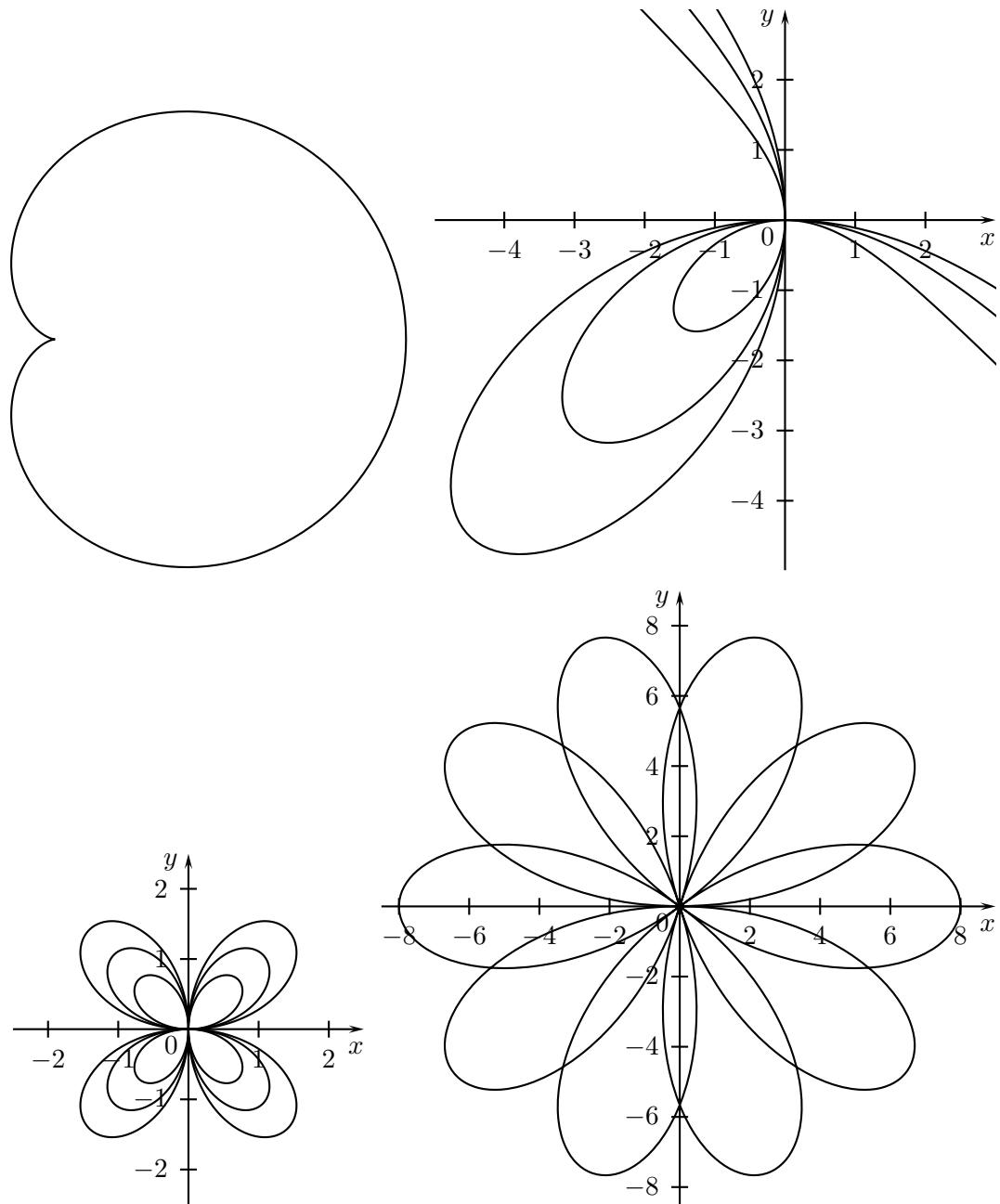
22          \pst@number\psyunit mul exch
23      } def}%
24
25 \else
26   \addto@pscode{%
27     \psplot@init
28     /x #1 def
29     /x1 #2 def
30     /dx x1 x sub \psk@plotpoints div def
31     /xy {
32       x \pst@number\psxunit mul
33       #3 \pst@number\psyunit mul
34     } def}%
35 \fi
36 \gdef\psplot@init{}%
37 \@pstfalse
38 \nameuse{testqp@\psplotstyle}%
39 \if@pst
40   \psplot@ii
41 \else
42   \psplot@iii
43 \fi
44 \endgroup
45 \ignorespaces}
46 \psset{plotpoints=200}%
47
48 \pspicture(-0.6,-3.3)(5,3.3)
49 % Cardioide
50 \psplot[polarplot=true,unit=0.5]{0}{360}{x cos 1 add 5 mul}
51 \endpspicture
52 \hfill
53 \pspicture*(-5,-5)(3,3)
54   \psaxes[arrowlength=2]{>}(0,0)(-4.99,-4.99)(3,3)
55   \rput[Br](3,-0.35){$x\$}
56   \rput[tr](-0.15,3){$y\$}
57   \rput[Br](-0.15,-0.35){$0\$}
58   \psset{polarplot=true}%
59   \psplot{140}{310}{3 neg x sin mul x cos mul
60           x sin 3 exp x cos 3 exp add div}
61   \psplot{140}{310}{6 neg x sin mul x cos mul
62           x sin 3 exp x cos 3 exp add div}
63   \psplot{140}{310}{9 neg x sin mul x cos mul
64           x sin 3 exp x cos 3 exp add div}
65 \endpspicture
66
67 \pspicture(-2.5,-2.5)(2.5,2.5)
68   \psaxes[arrowlength=2]{>}(0,0)(-2.5,-2.5)(2.5,2.5)
69   \rput[Br](2.5,-0.35){$x\$}
70   \rput[tr](-0.15,2.5){$y\$}
71   \rput[Br](-0.15,-0.35){$0\$}
72   \psset{plotstyle=curve,polarplot=true}%
73   \psplot{0}{360}{x cos 2 mul x sin mul}
74   \psplot{0}{360}{x cos 3 mul x sin mul}

```

```

75 \psplot{0}{360}{x cos 4 mul x sin mul}
76 \endpspicture
77 \hfill
78 \psset{unit=0.5}%
79 \pspicture(-8.5,-8.5)(9,9)
80 \psaxes[Dx=2,dx=2,Dy=2,dy=2,arrowlength=2]{->}(0,0)(-8.5,-8.5)(9,9)
81 \rput[Br](9,-0.7){$x$}
82 \rput[tr](-0.3,9){$y$}
83 \rput[Br](-0.3,-0.7){$0$}
84 \psplot[plotstyle=curve,polarplot=true]{0}{720}{8 2.5 x mul sin mul}
85 \endpspicture

```

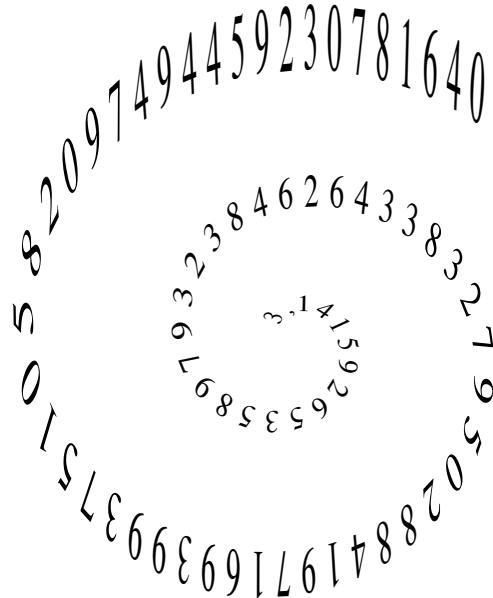


Another example, using both a polar plot and putting a text around it, with the **\pstextpath** macro. As we also want to slightly increase the size of the characters along the ellipse, we use a generic macro from Juergen Schlegelmilch to get one character at a time and apply an action on it).

```

1 % A \DoPerCharacter macro adapted from Juergen Schlegelmilch
2 % (posted on comp.text.tex - January 1998)
3 \def\DoPerCharacter#1#2#3{\nil{%
4 #1#2%
5 \edef\@tempa{#3}%
6 \ifx\@tempa\empty%
7 \else%
8   \DoPerCharacter#1#3\@nil%
9 \fi}%
10 %
11 \def\PerCharacter#1#2{\DoPerCharacter#1#2\@nil}%
12 %
13 \pst@dimh=0.7pt%
14 %
15 \def\CharacterAction#1{%
16 \scalebox{1}{\pst@number{\pst@dimh}}{#1}%
17 \advance\pst@dimh by 0.04pt}%
18 %
19 \pstextpath{\psplot*[linestyle=none,polarplot=true,plotpoints=300,unit=5]
20 {500}{-500}{1 2.7182818 x 200 div exp 1 add div}}% Spiral
21 {\PerCharacter{\CharacterAction}}%
22 {3.141592653589793238462643383279502884197169399375%
23 {10582097494459230781640}}

```



L.16 Non filled arrows

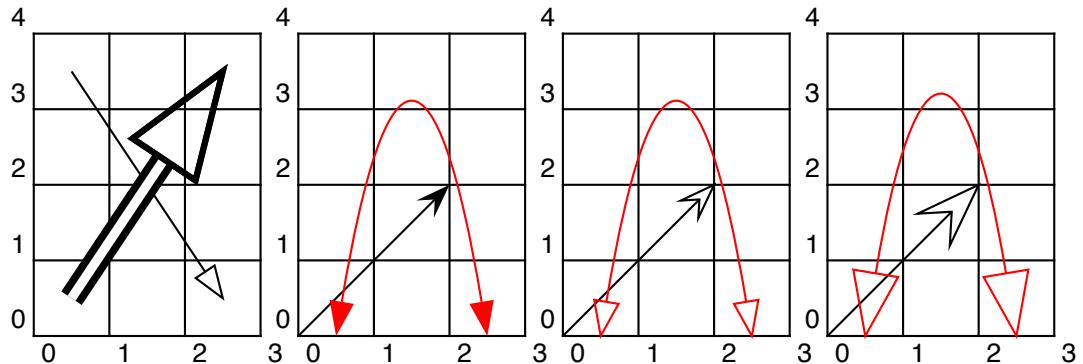
If we want to draw non filled arrows, we can define an `arrowfill` parameter (to have it with double arrows require to modify too the `psas@>>` and `psas@<<` macros.)

```
1 % D.G. - July 1999
2
3 % Redefinition of \psset@arrowscale to store the value
4 % of the X scale factor
5 \def\psset@arrowscale#1{%
6 \psset@arrowscale#1 \@nil
7 \pst@getscale{#1}\psk@arrowscale}
8
9 \def\psset@arrowscale@i#1 #2 \@nil{\edef\pst@arrowscale{#1}}
10
11 \def\pst@arrowscale{1}% Default value
12
13 % New parameter \newif\ifpsarrowfill
14 \def\psset@arrowfill#1{\@nameuse{psarrowfill#1}}
15 \psset@arrowfill{true}
16
17 % Modification of the PostScript macro "Arrow" to choose to fill
18 % or not the arrow (it require to restore the current linewidth,
19 % despite of the scaling)
20 \pst@def{Arrow}{{%
21 CLW mul add dup 2 div
22 /w ED mul dup
23 /h ED mul
24 /a ED
25 { 0 h T 1 -1 scale } if
26 gsave
27 \ifpsarrowfill
28 \else
29 \pst@number{\pslinewidth} \pst@arrowscale\space div
30 \ifpsdoubleline\space 4 div\fi\space
31 SLW
32 \fi
33 w neg h moveto
34 0 0 L w h L w neg a neg rlineto
35 \ifpsarrowfill
36 fill
37 \else
38 closepath stroke
39 \fi
40 grestore
41 0 h a sub moveto}>
42
43 \psset{subgriddiv=0}%
44
45 \pspicture(3,4)\psgrid
46 \psset{arrowfill=false,arrows=->,arrowinset=0}%
47 \psline[arrowsize=0.3](0.5,3.5)(2.5,0.5)
```

```

48  \pnode(0.5,0.5){A}
49  \pnode(2.5,3.5){B}
50  \ncline[linewidth=0.1,arrowsize=1,doubleline=true]{A}{B}
51  \endpspicture
52  \hfill
53  \psset{subgriddiv=0,arrowscale=3}%
54  \pspicture(3,4)\psgrid
55  \psline{->}(2,2)
56  \parabola[linecolor=red,arrowinset=0]{<->}(0.5,0)(1.5,3)
57  \endpspicture
58  \hfill
59  \pspicture(3,4)\psgrid
60  \psset{arrowfill=false}%
61  \psline{->}(2,2)
62  \parabola[linecolor=red,arrowinset=0]{<->}(0.5,0)(1.5,3)
63  \endpspicture
64  \hfill
65  \pspicture(3,4)\psgrid
66  \psset{arrowfill=false,arrowsize=0.2}%
67  \psline{->}(2,2)
68  \parabola[linecolor=red,arrowinset=0]{<->}(0.5,0)(1.5,3)
69  \endpspicture

```



L.17 Lines under a curve

Rather than joining the points of a curve, we can draw for each of these points a vertical line to the horizontal axis. We can still show the points of the curve, with the **showpoints** parameter.

```

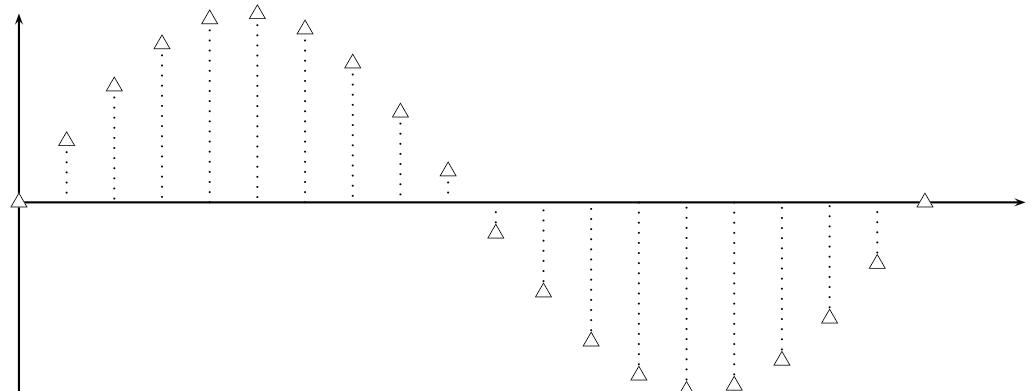
1 % D.G. - January 2000
2
3 \let\beginplot@LineToXAxis\beginplot@line
4 \def\endplot@LineToXAxis{\psLineToXAxis@ii}
5 \let\beginqp@LineToXAxis\beginqp@line
6 \let\doqp@LineToXAxis\doqp@line
7 \let\endqp@LineToXAxis\endqp@line
8 \let\testqp@LineToXAxis\testqp@line
9
10 \def\psLineToXAxis@ii{%
11 \addto@pscode{\pst@cp \psline@iii \tx@LineToXAxis}%

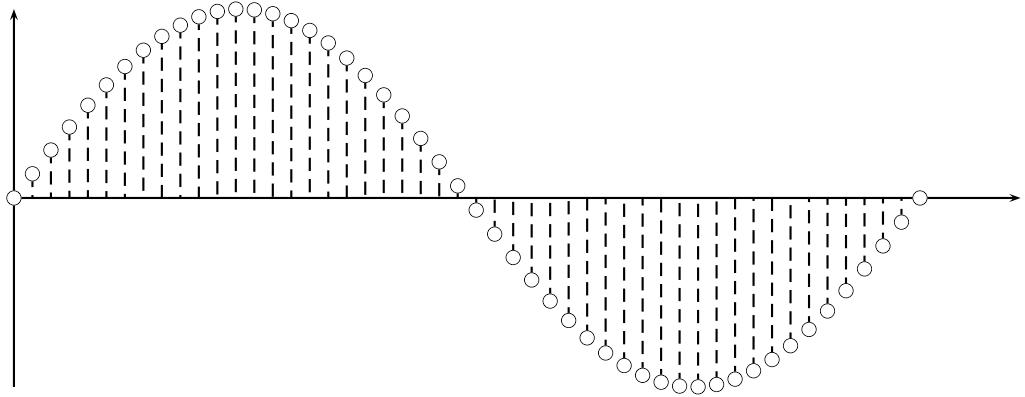
```

```

12 \end{OpenObj}
13
14 \def\tx@LineToXAxis{LineToXAxis }
15
16 % Adapted from the Line PostScript macro
17 \pst@def{LineToXAxis}{%
18 NArray
19 n 0 eq not
20 { n 1 eq { 0 0 /n 2 def } if
21 ArrowA
22 /n n 2 sub def
23 CP 2 copy moveto pop 0 Lineto
24 n { 2 copy moveto pop 0 Lineto } repeat
25 CP
26 4 2 roll
27 ArrowB
28 2 copy moveto pop 0
29 L
30 pop pop } if}>
31
32 \psset{xunit=0.0333,yunit=2.5}
33
34 \pspicture(0,-1)(400,1)
35 \psline{->}(0,0)(400,0)
36 \psline{->}(0,-1)(0,1)
37 \psplot[plotstyle=LineToXAxis,plotpoints=20,linestyle=dotted,
38 showpoints=true,dotstyle=triangle,dotsize=0.2]{0}{360}{x sin}
39 \endpspicture
40
41 \pspicture(0,-1)(400,1)
42 \psline{->}(0,0)(400,0)
43 \psline{->}(0,-1)(0,1)
44 \psplot[plotstyle=LineToXAxis,plotpoints=50,linestyle=dashed,
45 showpoints=true,dotstyle=o,dotsize=0.2]{0}{360}{x sin}
46 \endpspicture

```





L.18 Dashed lines

We can extend the dashed **linestyle** to accept dashes defined by two different segments.

```

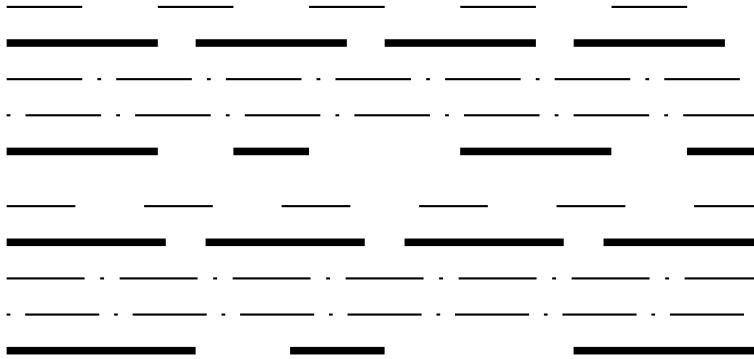
1 % D.G. - November 1997
2
3 \def\psset@dash#1{%
4 \pst@expandafter\psset@@dash{#1} {z@} {z@}
5   {\pst@missing} {\pst@missing} {}\\@nil
6
7 \edef\psk@dash{%
8 \pst@number\pst@dimg \pst@number\pst@dimh
9 \pst@number\pst@dimc \pst@number\pst@dimd}}
10
11 \def\psset@@dash#1 #2 #3 #4 #5@nil{%
12 \pssetlength{\pst@dimg}{#1}%
13 \pssetlength{\pst@dimh}{#2}%
14 \pssetlength{\pst@dimc}{#3}%
15 \pssetlength{\pst@dimd}{#4}}
16
17 \psset@dash{5pt 3pt}% Default value
18
19 \pst@def{DashLine}<%
20 dup 0 gt
21   { /a .5 def PathLength exch div }
22   { pop /a 1 def PathLength }
23 ifelse
24 /b ED
25 /x1 ED
26 /y1 ED
27 /x ED
28 /y ED
29 /z y x add y1 add x1 add def
30 /Coef b a .5 sub 2 mul y mul sub z Div round
31   z mul a .5 sub 2 mul y mul add b exch Div def
32 /y y Coef mul def
33 /x x Coef mul def
34 /y1 y1 Coef mul def

```

```

35 /x1 x1 Coef mul def
36 x1 0 gt y1 0 gt x 0 gt y 0 gt and
37 { [ y x y1 x1 ] 1 a sub y mul}
38 { [ 1 0] 0 }
39 ifelse
40 setdash
41 stroke>
42
43 \def\defTestDashedLine[#1]{%
44 \pspicture(0,-0.08)(10,0.08)
45 \psline[#1](10,0)
46 \endpspicture}
47
48 \psset{dashadjust=false}%
49 \TestDashedLine[linestyle=dashed,dash=1 1]
50 \TestDashedLine[linestyle=dashed,linewidth=1mm,dash=2 0.5]
51 \TestDashedLine[linestyle=dashed,dash=1 0.2 0.05 0.2]
52 \TestDashedLine[linestyle=dashed,dash=0.05 0.2 1 0.2]
53 \TestDashedLine[linestyle=dashed,linewidth=1mm,dash=2 1 1 2]
54
55 \psset{dashadjust=true}%
56 \TestDashedLine[linestyle=dashed,dash=1 1]
57 \TestDashedLine[linestyle=dashed,linewidth=1mm,dash=2 0.5]
58 \TestDashedLine[linestyle=dashed,dash=1 0.2 0.05 0.2]
59 \TestDashedLine[linestyle=dashed,dash=0.05 0.2 1 0.2]
60 \TestDashedLine[linestyle=dashed,linewidth=1mm,dash=2 1 1 2]

```



M Troubleshooting

DG: To complete and
recheck.

M.1 Compatibility problem between PStricks and the ‘color’ and ‘graphicx’ L^AT_EX packages

The ‘color’ [7] L^AT_EX package (also usable in PLAIN T_EX) is the way to use to define and manage colors in L^AT_EX and PLAIN T_EX since the arrival of L^AT_EX2e in 1993. Nevertheless, it is not strictly compatible with the color macros defined in PStricks, so, to be able to use the ‘color’ package, the

‘pstcol’ [65] one, also written by David Carlisle, must be used in place of both ‘color’ and PStricks (the core pstricks.sty / pstricks.tex file).

Another pitfall is in the usage of the ‘graphicx’ [27] L^AT_EX package (or the ‘graphics’ one), which, even if the ‘color’ package is not used, require that the ‘pstcol’ one has been loaded before.

In one word: with L^AT_EX always use the ‘pstcol’ package.

M.2 Compatibility problem between PStricks and the ‘graphicx’ L^AT_EX package for the \scalebox macro

Both PStricks and the ‘graphicx’ [27] L^AT_EX package define the \scalebox macro, but with a different syntax. si, the syntax to use depend of the order of loading of the PStricks (that is to say, as explained in the previous paragraph, ‘pstcol’) and ‘graphicx’.

M.3 Compatibility problem between the packages ‘fp’ and ‘multido’

The ‘fp’ [19] and ‘multido’ [45] packages both define the \FP@add and \FP@sub macros, which can lead to a clash if both are used in the same file (in fact, when using in a \multido loop an index starting with a \n... or \N.... The following patch allow to solve this problem.

```
1 % D.G. - March 2002 and July 2003
2
3 \usepackage{multido}
4
5 % Compatibility clash between 'fp' and 'multido'
6 % when using \n.. multido indexes
7 % (to add after 'multido' and before 'fp')
8 \let\FPaddMultiDo\FPadd
9 %
10 \def\FPsub#1#2{%
11   \edef\multido@temp{\noexpand\FPsub@#2\noexpand\@empty}%
12   \FPaddMultiDo{#1}{\multido@temp}%
13 %
14 \let\FPsubMultiDo\FPsub
15 %
16 \def\multido@init@n#1#2#3{%
17   \edef#3{#1}%
18   \ifnum\multido@count<\z@
19     \expandafter\FPsubMultiDo
20   \else
21     \expandafter\FPaddMultiDo
22 \fi}
```

```
23 {0}{#2}\multido@temp
24 \multido@addtostep\do\FPaddMultiDo{\do#3}{\multido@temp}{\do#3}}
25
26 \usepackage[nomessages]{fp}
```

XIV Thanks

All the credit for the initiative of merging the different parts of the documentation from the years 1993 and 1994, integrating and repackaging all the files, is due to Rolf Niepraschk <Rolf.Niepraschk@ptb.de>, who started this work at the end of 2002.

Special thanks are due to Manuel Luque <MLuque5130@aol.com> who wrote nearly all the chapter X on the ‘pst-3d’ package, is one of the authors of the ‘pst-labo’ package used in the chapter XII, give an example used in the *Animated graphics* section B and sent various remarks and suggestions at the different steps of work in the process of repackaging this documentation.

Thanks are also due to the persons who correct the English of the new sections and proofread them (alphabetical order):

- John Frampton <jframpto@lynx.dac.neu.edu>,
- Manjusha Joshi <manjushasj@math.unipune.ernet.in>,
- Anthony Tekatch <anthony@unihedron.com>.

XV References

- [1] Acrobat. Adobe Systems Incorporated. See
<http://www.adobe.com/products/acrobat>
- [2] AlDraTeX. A high-level programming language for drawing figures, by Eitan M. Gurari [CTAN:graphics/dratex](#) See also
<http://www.cis.ohio-state.edu/~gurari/systems.html>
- [3] Guillaume Appolinaire, *Calligrammes*, Gallimard, 1925 (in French).
- [4] ‘arrayjob’. Management of arrays in TeX, by Zuhuan Jiang.
[CTAN:macros/generic/arrayjob](#)
- [5] Bakoma TeX. TeX implementation, by Basil K. Malyshev (Windows systems only). [CTAN:nonfree/systems/win32/bakoma](#) See also
<http://www.tex.ac.uk/tex-archive/systems/win32/bakoma>
- [6] Emmanuel Chailloux, Guy Cousineau and André Suárez, *Programmation fonctionnelle de graphismes pour la production d’illustrations techniques*, Technique et science informatique, Volume 15, Number 7, pages 977–1007, 1996 (in French).
- [7] ‘color’. The color package, LATEX Colour interface, by David Carlisle.
[CTAN:macros/latex/graphics/color.dtx](#)
- [8] ConTeXt. A general purpose TeX macro package.
[CTAN:macros/context](#) See also
<http://www.pragma-ade.nl/context.htm>
- [9] André Deledicq, *Le monde des pavages*, ACL Éditions, 1997 (in French).
- [10] Dia. Interactive diagram editor. See
<http://www.lysator.liu.se/~alla/dia>
- [11] ‘docstrip’. The docstrip package, by Frank Mittelbach, Denys Duchier, Johannes Braams, Marcin Woliński and Mark Wooding.
[CTAN:macros/latex/base/docstrip.dtx](#)
- [12] Victor Eijkhout, *TeX by Topic*, Addison-Wesley, 1992.
- [13] Philippe Esperet and Denis Girou, *Coloriage du pavage dit de Truchet*, Cahiers GUTenberg, Number 31, pages 5–18, Décembre 1998 (in French). Available at
<http://www.gutenberg.eu.org/pub/GUTenberg/publicationsPDF/31-girou.pdf>

- [14] Eukleides. Euclidean geometry drawing tool, by Christian Obrecht.
See <http://perso.wanadoo.fr/obrecht>
- [15] ‘fancybox’. Box tips and tricks for L^AT_EX, by Timothy van Zandt.
<CTAN:macros/latex/contrib/fancybox>
- [16] ‘fancyvrb’. Fancy Verbatims in L^AT_EX, by Timothy van Zandt.
<CTAN:macros/latex/contrib/fancyvrb>
- [17] Gabriel Valiente Feruglio, *Typesetting commutative diagrams*, TUG-Boat, Volume 15, Number 4, pages 466–484, 1994.
- [18] ‘fltpoint’. Floating point arithmetic with T_EX, by Eckhart Guthörlrhein.
<CTAN:macros/latex/contrib/fltpoint>
- [19] ‘fp’. Fixed point arithmetic for T_EX, by Michael Mehlich.
<CTAN:macros/latex/contrib/fp>
- [20] GhostScript. PostScript interpreter.
See <http://www.cs.wisc.edu/~ghost> and
<http://sourceforge.net/projects/ghostscript>
- [21] gifsicle. Generation of gif animated graphic files, by Eddie Kohler.
See <http://www.lcdf.org/~eddietwo/gifsicle>
- [22] Denis Girou, *Présentation de PStricks*, Cahiers GUTenberg, Number 16, pages 21–70, February 1994 (in French). Available at <http://www.gutenberg.eu.org/pub/GUTenberg/publicationsPDF/16-girou.pdf>
- [23] Denis Girou, *Building high level objects in PStricks*, TUG’95, St Petersburg, Florida, 1995. Available at http://www.tug.org/applications/PStricks/TUG95-PStricks_4.ps.gz
- [24] gnuplot. Interactive function plotting program. See <http://www.gnuplot.info>
- [25] Michel Goossens, Sebastian Rahtz and Frank Mittelbach, *The L^AT_EX Graphics Companion*, Addison-Wesley, 1997.
- [26] GraphicsMagick. Collection of tools and libraries to read, write, and manipulate images in various formats (formed as a branch from ImageMagick 5.5.2). See <http://www.graphicsmagick.org/>
- [27] ‘graphicx’. The `graphicx` package, Enhanced L^AT_EX Graphics, by David Carlisle and Sebastian Rahtz.
<CTAN:macros/latex/graphics/graphicx.dtx>
- [28] Branko Grünbaum and Geoffrey Shephard. *Tilings and Patterns*, Free-man and Company, 1987.
- [29] Eitan M. Gurari, *T_EX and L^AT_EX: Drawing and Literate Programming*, McGraw-Hill, 1994.
- [30] Alan Hoenig, *T_EX Unbound: L^AT_EX & T_EX Strategies, Fonts, Graphics, and More*, Oxford University Press, 1998.
- [31] ‘ifthen’. The `ifthen` package, by David Carlisle.
<CTAN:macros/latex/base/ifthen.dtx>

- [32] ImageMagick. Collection of tools and libraries to read, write, and manipulate images in various formats. See <http://wwwimagemagick.org/>
- [33] Laura E. Jackson and Herbert Voß, *Die Plot-Funktionen von PostScript*, Die TeXnische Komödie, Volume 2/02, pages 27–34, June 2002 (in German).
- [34] jPicEdt. Interactive picture editor, by Sylvain Reynal. See <http://www.picedt.org>
- [35] ‘keyval’. key=value parser, by David Carlisle. <CTAN:macros/latex/required/graphics/keyval.dtx>
- [36] Peter Kleiweg, PostScript code to create a lighten effect on characters. See the file `depth.ps` at <http://odur.let.rug.nl/~kleiweg/postscript/postscript.html>
- [37] Donald E. Knuth, *Computers and Typesetting*, Vol. A, *The T_EXbook*, Addison-Wesley, 1986.
- [38] Benoit B. Mandelbrot, *The Fractal Geometry of Nature*, Freeman, 1977.
- [39] Henry McGilton and Mary Campione, *PostScript by Example*, Addison-Wesley, 1992.
- [40] MetaFun. MetaPost macros for ConTeXt, by Hans Hagen. See <http://www.pragma-ade.nl/metapost.htm> and <http://www.pragma-ade.nl/general/manuals/metafun-p.pdf>
- [41] MetaObj. Very High-Level Objects in MetaPost, by Denis Roegel. See <CTAN:graphics/metapost/contrib/macros/metaobj>
- [42] MetaPost. A graphics language based on Knuth’s MetaFont, by John D. Hobby. See <CTAN:graphics/metapost> and <http://cm.bell-labs.com/who/hobby/MetaPost.html>
- [43] MFpic. A drawing environment for T_EX and L^AT_EX, generating Metafont or MetaPost commands, by Dan Luecking. <CTAN:graphics/mfpic>
See also <http://comp.uark.edu/~luecking/tex/mfpic.html>
- [44] MNG. Multiple-image Network Graphics format. See <http://www.libpng.org/pub/mng/> and <http://www.libpng.org/pub/png/png-sitemap.html#animation>
- [45] ‘multido’. A loop macro for Generic T_EX, by Timothy van Zandt. <CTAN:macros/generic/multido>
- [46] Portable Document Format Reference, by Adobe Systems Incorporated. See <http://partners.adobe.com/asn/tech/pdf/specifications.jsp>
- [47] pdfT_EX. Variant of the T_EX program, by H[ा]n Th^é Thành. See <http://tug.org/applications/pdftex>

- [48] pdfTricks. PSTricks support in PDF with pdf \TeX , by Radhakrishnan CV, Rajagopal CV and Antoine Chambert-Loir. [CTAN:macros/latex/contrib/pdftricks](#)
- [49] Heinz-Otto Peitgen, Hartmut Jürgens and Dietmar Saupe, *Chaos and Fractals – New Frontiers of Science*, Springer Verlag, 1992.
- [50] *PostScript Language Tutorial and Cookbook*, Adobe Systems Incorporated, Addison-Wesley, 1985. See <http://partners.adobe.com/asn/tech/ps/index.jsp>
- [51] *PostScript Language Reference Manual*, Adobe Systems Incorporated, Addison-Wesley, 3 edition, 1999.
- [52] ‘preview’. Extraction of previews from \TeX documents, by David Kastrup. [CTAN:macros/latex/contrib/preview](#) See <http://preview-latex.sourceforge.net>
- [53] ‘ps4pdf’. Usage of PostScript commands inside a pdf \TeX processed document, by Rolf Niepraschk. [CTAN:macros/latex/contrib/ps4pdf](#)
- [54] ‘pst-3dplot’. PSTricks contribution package for 3d plots, by Herbert Voß. [CTAN:graphics/pstricks/contrib/pst-3dplot](#) See also <http://www.perce.de/LaTeX/pst-3dplot>
- [55] ‘pst-blur’. PSTricks contribution package for blurred shadows, by Martin Giese. [CTAN:graphics/pstricks/contrib/pst-blur](#)
- [56] ‘pst-csyn’. PSTricks contribution package for color synthesis of overlapped surfaces, by Denis Girou and Manuel Luque. [CTAN:graphics/pstricks/contrib/pst-csyn](#)
- [57] ‘pst-chrt’. PSTricks contribution package for business charts, by Denis Girou (unreleased).
- [58] ‘pst-circ’. PSTricks contribution package for electric circuits, by Christophe Jorssen and Herbert Voß. [CTAN:graphics/pstricks/contrib/pst-circ](#) See also <http://pstcirc.free.fr>
- [59] ‘pst-eucl’. PSTricks contribution package for Euclidean geometry, by Dominique Rodriguez. <http://dominique.rodriguez.9online.fr/pst-eucl>
- [60] ‘pst-lens’. PSTricks contribution package for lens, by Denis Girou and Manuel Luque. [CTAN:graphics/pstricks/contrib/pst-lens](#)
- [61] ‘pst-labo’. PSTricks contribution package for chemistry experiment drawings, by Manuel Luque and Christophe Jorssen.
- [62] ‘pst-li3d’. PSTricks contribution package for three dimensional lighten effect on characters and PSTricks graphics, by Denis Girou. [CTAN:graphics/pstricks/contrib/pst-li3d](#)
- [63] ‘pst-poly’. PSTricks contribution package for polygons, by Denis Girou. [CTAN:graphics/pstricks/contrib/pst-poly](#)

- [64] ‘pst-slpe’. PSTricks contribution package for improved gradients, by Martin Giese. [CTAN:graphics/pstricks/contrib/pst-slpe](#)
- [65] ‘pstcol’. The `pstcol` package, PSTricks color compatibility, by David Carlisle. [CTAN:macros/latex/graphics/pstcol.dtx](#)
- [66] PStill. PS/EPS to PDF converter, by Frank Siegert. See <http://www.pstill.com>
- [67] PSTricks tutorial, by Alex AJ, CV Radhakrishnan and E. Krishnan (for the India T_EX Users Group) See <http://sarovar.org/projects/pstricks> and <http://www.tug.org.in/tutorial/pstricks>
- [68] PSTricks Web pages, maintained by Denis Girou. See <http://www.tug.org/applications/PSTricks>
- [69] PSTricks Web pages, maintained by Manuel Luque (in French). See <http://members.aol.com/Mluque5130>
- [70] PSTricks Web pages on the Syracuse site, maintained by Jean-Michel Sarlat (in French). See <http://melusine.eu.org/syracuse/pstricks>
- [71] PSTricks Web pages, maintained by Herbert Voß. See <http://www.pstricks.de>
- [72] ‘random’. Generating “random” numbers in T_EX, by Donald Arseneau. [CTAN:macros/generic/random.tex](#)
- [73] ‘realcalc’. Real arithmetic with big values and high precision, by Frank Buchholz. [CTAN:macros/generic/realcalc](#)
- [74] ‘repeat’. Repeat loop macro, by Victor Eijkhout [CTAN:macros/generic/eijkhout/repeat.tex](#)
- [75] *Using Imported Graphics in L^AT_EX2e*, by Keith Reckdahl, December 1997, available on [CTAN:info/epslatex.pdf](#)
- [76] Denis Roegel, *MetaObj: Very High-Level Objects in MetaPost*, TUG-Boat, Volume 23, Number 1, pages 93–100, 2002.
- [77] David Salomon, *The Advanced T_EXbook*, Springer-Verlag, 1995.
- [78] ‘Seminar’. A L^AT_EX style for slides and notes, by Timothy van Zandt, April 1993. [CTAN:macros/latex/contrib/seminar](#) See also <http://www.tug.org/applications/Seminar>
- [79] Timothy van Zandt and Denis Girou, *Inside PSTricks*, TUGBoat, Volume 15, Number 3, pages 239–246, 1994.
- [80] ‘Vaucanson-G’. PSTricks contribution package for drawing automata and graphs (with L^AT_EX), by Sylvain Lombardy and Sylvain Lombardy. See <http://www.liafa.jussieu.fr/~lombardy/Vaucanson-G>
- [81] Herbert Voß and Laura E. Jackson, *Die mathematischen Funktionen von PostScript*, Die TeXnische Komödie, Volume 1/02, pages 40–47, March 2002 (in german).

- [82] VT_EX. T_EX implementation, by MicroPress Incorporated (Linux, Windows and some other systems). [CTAN:systems/vtex](#) See also <http://www.micropress-inc.com> and <http://www.micropress-inc.com/linux.htm>
- [83] Xy-pic. Typesetting graphs and diagrams in T_EX, by Kristoffer H. Rose and Ross Moore. [CTAN:macros/latex/contrib/xy-pic](#) See also <http://www.tug.org/applications/Xy-pic>
- [84] Hao Wang, *Games, Logic and Computers*, Scientific American, pages 98–106, 1965.