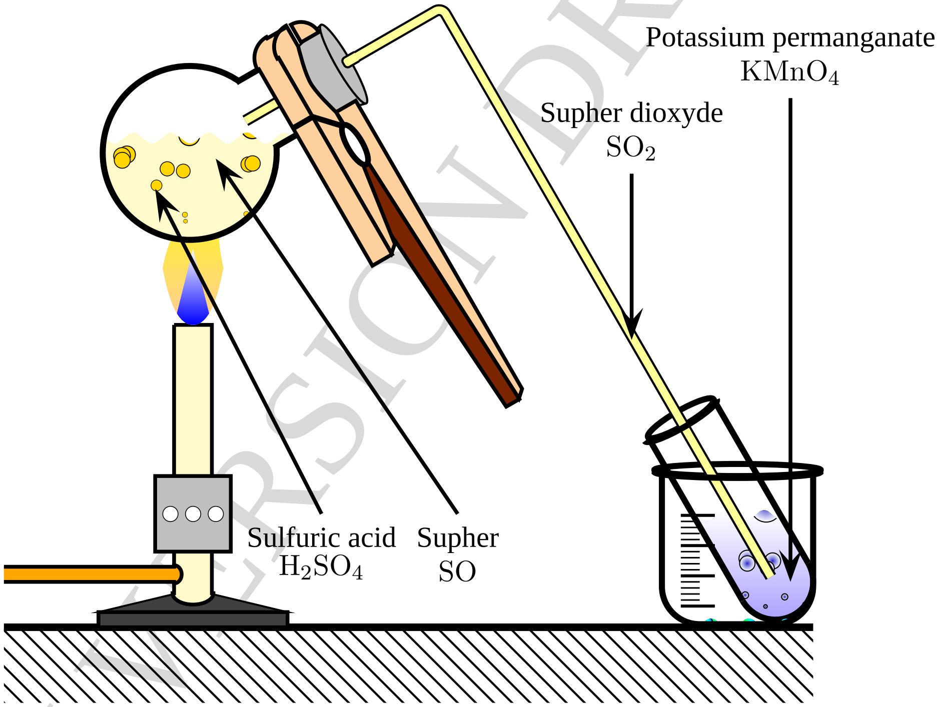
PSTricks

PostScript macros for Generic TeX

User’s Guide[1]

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>

---

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

Contents

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 ConTeXt, 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 ConTeXt and should work with other TeX macro packages as well.

pstricks

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**.

I

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 \lput.
```

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

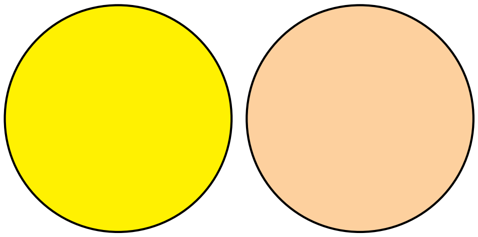
black, darkgray, gray, lightgray, and white,

and the colors

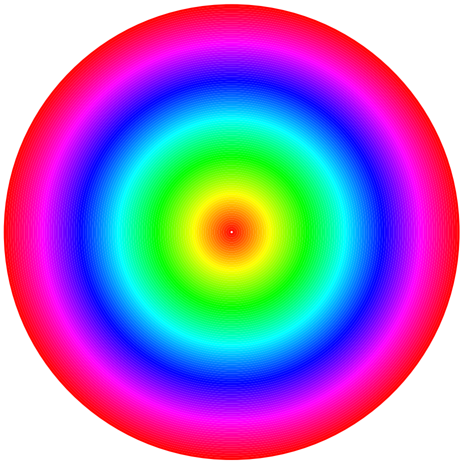
red, green, blue, cyan, magenta, and yellow

are predefined in PSTricks.

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



```
\psset{fillstyle=solid}%
\pscircle[fillcolor=yellow]{1.5}
\definecolor{LightOrange}  {cmyk}{0,0.2,0.4,0}%
\pscircle[fillcolor=LightOrange](3.2,0){1.5}
```



```
\psset{unit=3}%
\multido{\nHue=0.01+0.01}{100}{%
   \definecolor{MyColor}{hsb}{\nHue,1,1}%
   \pscircle[linewidth=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

```
\psset{fillcolor=yellow}
\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:

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

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

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

```
{\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.,

```
\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:

```
\psset{linewidth=.5cm}
\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 LaTeX parlance, a "length"), and *dim* is a length with optional unit. These are analogous to LaTeX's \setlength and \addtolength.

Coordinate pairs have the form (x,y). The origin of the coordinate system is at TeX'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:

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

```
\degrees[100]
```

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

```
\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* **Default: .8pt**
linecolor=*color* **Default: black**

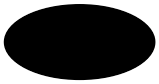
The **linewidth** is stored in the dimension register **\pslinewidth**, and the **linecolor** is stored in the command **\pslinecolor**.

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,

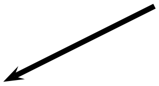


```
\psellipse*(1,.5)(1,.5)
```

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.,



```
\psline[linewidth=2pt]{<-}(2,1)
```

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.

---

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

II Basic graphics objects

6 Lines and polygons

The objects in this section also use the following parameters:

linearc=*dim* **Default: 0pt**

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.

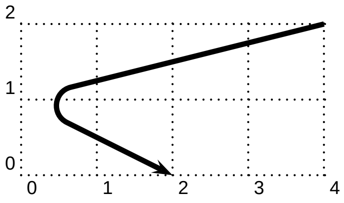
cornersize=*relative/absolute* **Default: relative**

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*)

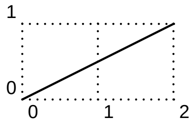
This draws a line through the list of coordinates. For example:



```
\psline[linewidth=2pt,linearc=.25]{->}(4,2)(0,1)(2,0)
```

\qline(*coor0*)(*coor1*)

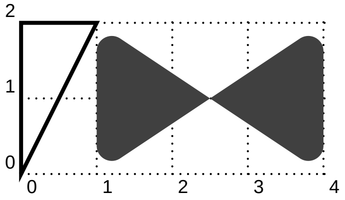
This is a streamlined version of **\psline** that does not pay attention to the **arrows** parameter, and that can only draw a single line segment. Note that both coordinates are obligatory, and there is no optional argument for setting parameters (use **\psset** if you need to change the **linewidth**, or whatever). For example:



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

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

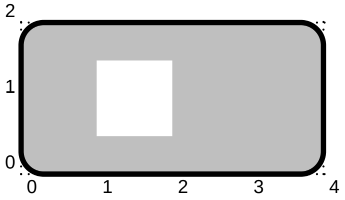
This is similar to **\psline**, but it draws a closed path. For example:



```
\pspolygon[linewidth=1.5pt](0,2)(1,2)
\pspolygon*[linearc=.2,linecolor=darkgray](1,0)(1,2)(4,0)(4,2)
```

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

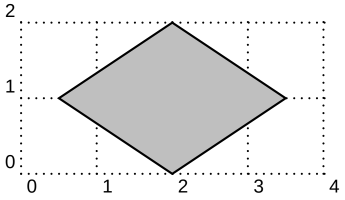
\psframe draws a rectangle with opposing corners (*x0*,*y0*) and (*x1*,*y1*). For example:



```
\psframe[linewidth=2pt,framearc=.3,fillstyle=solid,
    fillcolor=lightgray](4,2)
\psframe*[linecolor=white](1,.5)(2,1.5)
```

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

\psdiamond draws a diamond centered at (*x0*,*y0*), and with the half width and height equal to *x1* and *y1*, respectively.



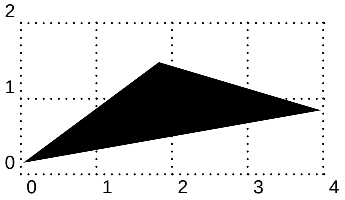
```
\psdiamond[framearc=.3,fillstyle=solid,
    fillcolor=lightgray](2,1)(1.5,1)
```

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 (*x0*,*y0*), and with width (base) and height equal to *x1* and *y1*, respectively.

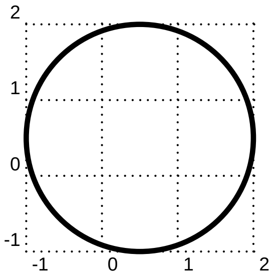


```
\pstriangle*[gangle=10](2,.5)(4,1)
```

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:



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

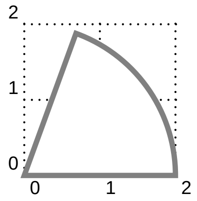
\qdisk(*coor*){*radius*}

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**:

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

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

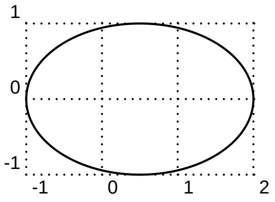
This draws a wedge whose center is at (*x0*,*y0*), that has radius *radius*, and that extends counterclockwise from *angle1* to *angle2*. The angles must be specified in degrees. For example:



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

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

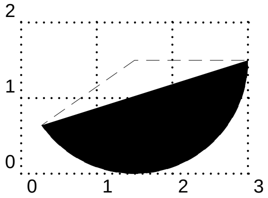
(*x0*,*y0*) is the center of the ellipse, and *x1* and *y1* are the horizontal and vertical radii, respectively. For example:



```
\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:



```
\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: 0pt**

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*.

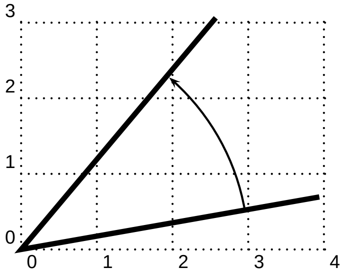
arcsepB=*dim* **Default: 0pt**

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:



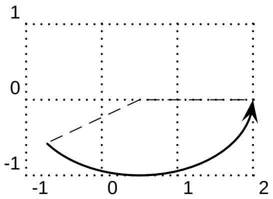
```
\SpecialCoor
\psline[linewidth=2pt](4;50)(0,0)(4;10)
\psarc[arcsepB=2pt]{->}{3}{10}{50}
```

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



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

---

[3] However, 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.

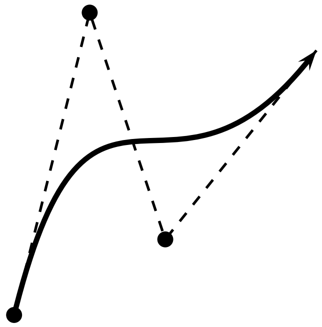
\psellipticarcn*[*par*]{*arrows*}(*x0*,*y0*)***(x1,y1){angleA}{angleB}***

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:



```
\psbezier[linewidth=2pt,showpoints=true]{->}(0,0)(1,4)(2,1)(4,3.5)
```

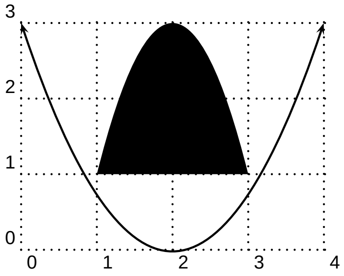
showpoints=true puts dots in all the control points, and connects them by dashed lines, which is useful when adjusting your bezier curve.

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

Starting at (*x0*,*y0*), **\parabola** draws the parabola that passes through (*x0*,*y0*) and whose maximum or minimum is (*x1*,*y1*). For example:

---

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



```
\parabola*(1,1)(2,3)
\psset{xunit=.01}
\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.

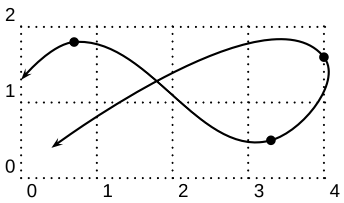
Lower values of *num2* tighten the curve where the angle ABC is greater than 45 degrees, and loosen the curve elsewhere.

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:

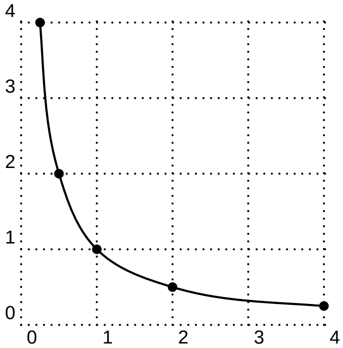


```
\pscurve[showpoints=true]{<->}(0,1.3)(0.7,1.8)
   (3.3,0.5)(4,1.6)(0.4,0.4)
```

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

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

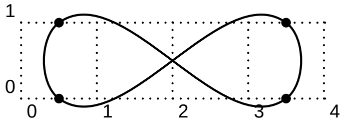
This is like **\pscurve**, but the curve is not extended to the first and last points. This gets around the problem of trying to determine how the curve should join the first and last points. The `e` has something to do with “endpoints”. For example:



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



```
\psccurve[showpoints=true]
    (.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:

Style	*Example*	*Style*	*Example*
*	● ● ● ● ●	square	□ □ □ □ □
o	○ ○ ○ ○ ○	Bsquare	□ □ □ □ □
Bo	○ ○ ○ ○ ○	square*	■ ■ ■ ■ ■
x	× × × × ×	diamond	◊ ◊ ◊ ◊ ◊
+	+ + + + +	Bdiamond	◊ ◊ ◊ ◊ ◊
B+	+ + + + +	diamond*	◆ ◆ ◆ ◆ ◆
asterisk	* * * * *	triangle	△ △ △ △ △
Basterisk	* * * * *	Btriangle	△ △ △ △ △
oplus	⊕ ⊕ ⊕ ⊕ ⊕	triangle*	▲ ▲ ▲ ▲ ▲
otimes	⊗ ⊗ ⊗ ⊗ ⊗	pentagon	⬠ ⬠ ⬠ ⬠ ⬠
|	| | | | |	Bpentagon	⬠ ⬠ ⬠ ⬠ ⬠
B|	| | | | |	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**

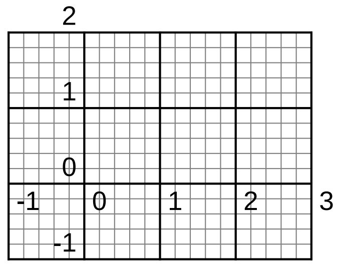
After setting the size and scaling the dots, the dots are rotated by *angle*.

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 (*x1*,*y1*) and (*x2*,*y2*). The intervals are numbered, with the numbers positioned at *x0* and *y0*. The coordinates are always interpreted as Cartesian coordinates. For example:



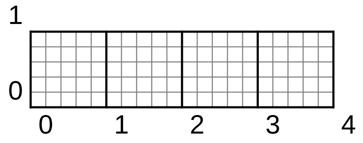
```
\psgrid(0,0)(-1,-1)(3,2)
```

(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.

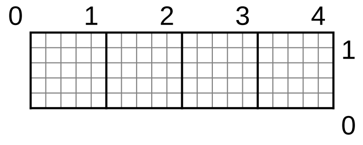
If the (*x0*,*y0*) coordinate is omitted, (*x1*,*y1*) is used. The default for (*x1*,*y1*) is (0,0). If you don't give any coordinates at all, then the coordinates of the current **\pspicture** environment are used or a 10x10 grid is drawn. Thus, you can include a **\psgrid** command without coordinates 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 *x1* (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



```
\psgrid(0,0)(4,1)
```

and



```
\psgrid(4,1)(0,0)
```

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.

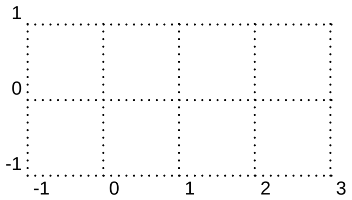
subgridcolor=*color* **Default: gray**

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:



```
\psgrid[subgriddiv=1,griddots=10,gridlabels=7pt](-1,-1)(3,1)
```

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

```
\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

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

11 Plots

pst-plot

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 %.

---

[5] For 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

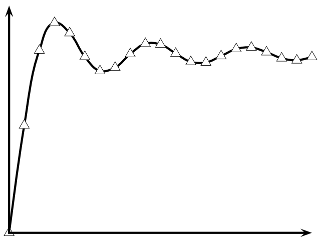
```
\readdata{\foo}{foo.data}
\readdata{\bar}{bar.data}
\dataplot{\foo\bar}
\dataplot[origin={0,1}]{\bar}
```

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 Integral(sin(x)). The data was generated by Mathematica, with

```
Table[{x,N[SinIntegral[x]]},{x,0,20}]
```

and then copied to this document.



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

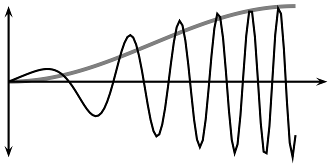
```
...
\listplot[plotstyle=curve,showpoints=true,
   dotstyle=triangle]{\mydata}
...
```

\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.,

```
\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 $\sin^2(x)$:

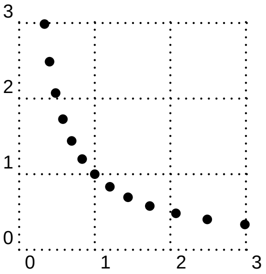


```
\psset{xunit=1.2pt}
\psplot[linecolor=gray,linewidth=1.5pt,plotstyle=curve]%
   {0}{90}{x  sin  dup  mul}
\psplot[plotpoints=100]{0}{90}{x  sin  x  2  div  2  exp  cos  mul}
\psline{<->}(0,-1)(0,1)
\psline{->}(100,0)
```

\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 $x(t)$ $y(t)$.

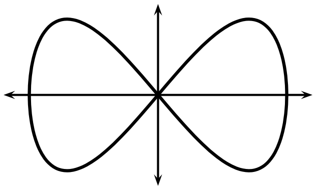
For example,



```
\parametricplot[plotstyle=dots,plotpoints=13]%
   {-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 $(\sin(t), \sin(2t))$:



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

III 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: 0pt,0pt**

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

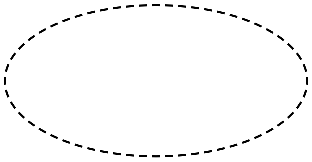
The following graphics parameters (in addition to **linewidth** and **linecolor**) determine how the lines are drawn, whether they be open or closed curves.

linestyle=*style* **Default: solid**

Valid styles are none, solid, dashed and dotted.

dash=*dim1 dim2* **Default: 5pt 3pt**

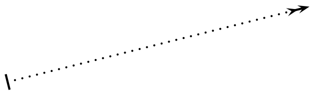
The *black-white* dash pattern for the dashed line style. For example:



```
\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



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

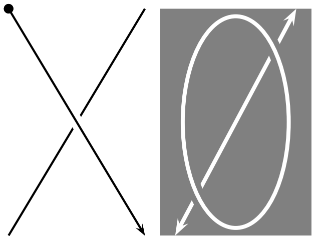
border=*dim* **Default: 0pt**

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:



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

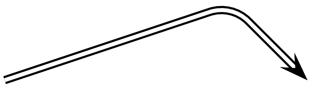
doublesep=*dim* **Default: 1.25\pslinewidth**

See **doubleline**, above.

doublecolor=*color* **Default: white**

See **doubleline**, above.

Here is an example of double lines:



```
\psline[doubleline=true,linearc=.5,
   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.

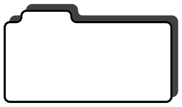
shadowangle=*angle* **Default: -45**

See **shadow**, above.

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

See **shadow**, above.

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

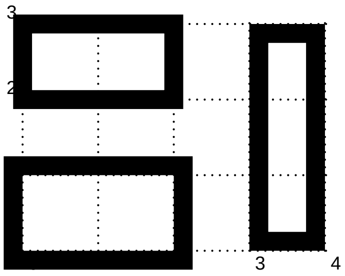


```
\pspolygon[linearc=2pt,shadow=true,shadowangle=45,
    xunit=1.1](-1,-.55)(-1,.5)(-.8,.5)(-.8,.65)
    (-.2,.65)(-.2,.5)(1,.5)(1,-.55)
```

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:



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

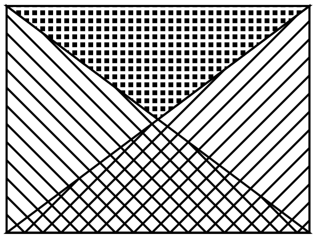
hatchcolor=*color* **Default: black**

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]



```
\pspolygon[fillstyle=vlines](0,0)(0,3)(4,0)
\pspolygon[fillstyle=hlines](0,0)(4,3)(4,0)
\pspolygon[fillstyle=crosshatch*,fillcolor=black,
   hatchcolor=white,hatchwidth=1.2pt,hatchsep=1.8pt,
   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.

---

[6] PSTricks 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.

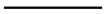
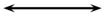
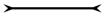
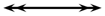
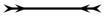
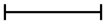
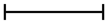
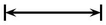
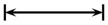
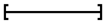
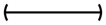
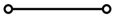
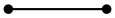
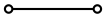
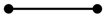
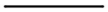
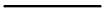
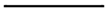
Each of the pure graphics objects (except those beginning with q) has a starred version that produces a solid object of color **linecolor**. (It automatically sets **linewidth** to zero, **fillcolor** to **linecolor**, **fillstyle** to solid, and **linestyle** to none.)

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]

Value	*Example*	*Name*
-		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.

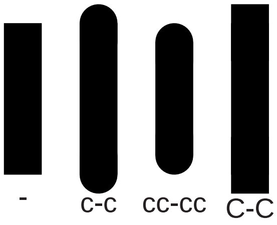
You can also mix and match. E.g., ->, *-) and [-> are all valid values of the **arrows** parameter.

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

---

[7] This 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:



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

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

you can write

```
\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**

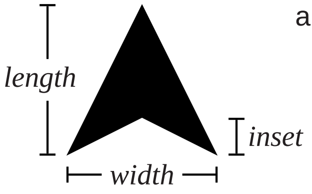
The width of arrowheads is *dim* plus *num* times **linewidth** (if the optional ‘num’ is inclued). See the diagram below.

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.



arrowsize = *dim num*
width = *num* × **linewidth** + *dim*
length = **arrowlength** × *width*
inset = **arrowinset** × *length*

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

```
\newpsobject{myline}{psline}{linecolor=green,linestyle=dotted}
\newpsobject{mygrid}{psgrid}{subgriddiv=1,griddots=10,
   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:

```
\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,

```
\newpsstyle{mystyle}{linecolor=green,linestyle=dotted}
\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 you 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 plinestyle 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:

Value	*Type of path*
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.

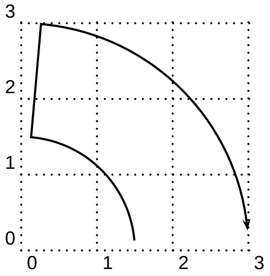
There are three types of graphics objects:

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

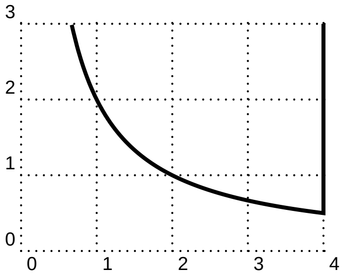


```
\pscustom{%
  \psarc(0,0){1.5}{5}{85}
  \psarcn{->}(0,0){3}{85}{5}}
```

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:

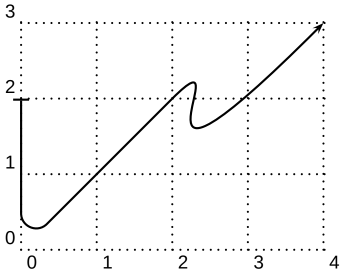


```
\pscustom[linewidth=1.5pt]{%
  \psplot[plotstyle=curve]{.67}{4}{2 x div}
  \psline(4,3)}
```

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

---

[8] The 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.



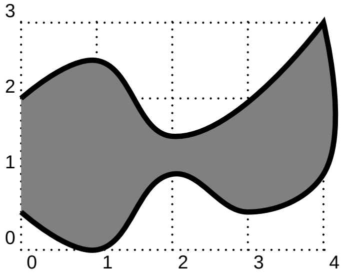
```
\pscustom{%
    \psline[linearc=.2]{|-}(0,2)(0,0)(2,2)
    \psbezier{->}(2,2)(3,3)(1,0)(4,3)}
```

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

liftpen=*0/1/2* **Default: 0**

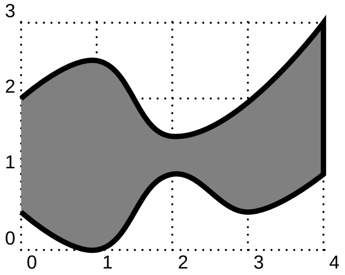
parameter.

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



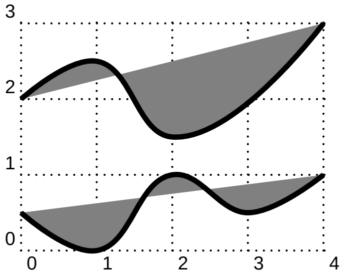
```
\pscustom[linewidth=2pt,fillstyle=solid,fillcolor=gray]{%
    \pscurve(0,2)(1,2.5)(2,1.5)(4,3)
    \pscurve(4,1)(3,0.5)(2,1)(1,0)(0,.5)}
```

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:



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



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

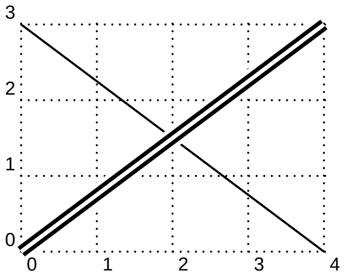
This moves the current point to (x,y).

\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):



```
\psline(0,3)(4,0)
\pscustom[linecolor=white,linewidth=1.5pt]{%
  \psline(0,0)(4,3)
  \stroke[linewidth=5\pslinewidth]
  \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 have nested **\gsave**-**\grestore** pairs.

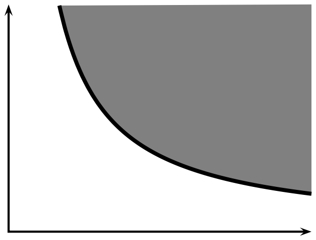
---

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

\grestore

See above.

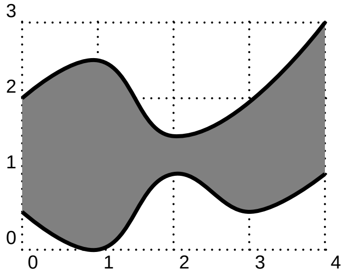
Here is an example that fixes an earlier example, using **\gsave** and **\grestore**:



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



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

```
\rotate{-90}
\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 TeX 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 componments 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(*x1*,*y1*)(*x2*,*y2*)(*x3*,*y3*)

This is like **\curveto**, but (*x1*,*y1*), (*x2*,*y2*) and (*x3*,*y3*) 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(*x1*,*y1*)(*x2*,*y2*)...(*xn*,*yn*)

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

\rcoor(*x1*,*y1*)(*x2*,*y2*)...(*xn*,*yn*)

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*}

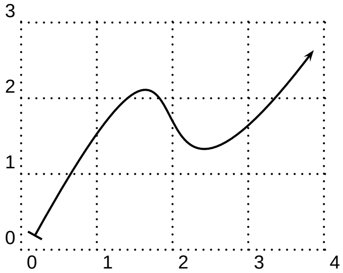
This defines the PostScript operators ArrowA and ArrowB so that

```
x2 y2 x1 y1 ArrowA
x2 y2 x1 y1 ArrowB
```

each draws an arrow(head) with the tip at (*x1*,*y1*) and pointing from (*x2*,*y2*). ArrowA leaves the current point at end of the arrowhead, where a connect line should start, and leaves (*x2*,*y2*) on the stack. ArrowB does not change the current point, but leaves

```
x2 y2 x1’ y1’
```

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:



```
\pscustom{
   \arrows{|->}
   \code{
      80  140  5  5  ArrowA
      30  -30  110  75  ArrowB
      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 (*x0*,*y0*) (by default, (0,0)) and whose upper right-hand corner is at (*x1*,*y1*).

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]

---

[10] LaTeX users can instead write:

\begin{pspicture} *stuff* \end{pspicture}

[11] When 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 **\pstverbscale**.

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>}
```

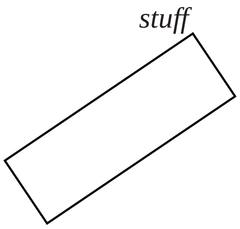
With the optional * argument, *stuff* is first put in a

```
\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 arguments 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,



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

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

Letter	*Short for*	*Equiv. to*	*Letter*	*Short for*	*Equiv. to*
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 a two of the PSTricks put commands. The most basic one command is

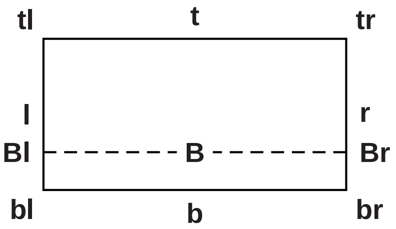
\rput*[*refpoint*]{*rotation*}**(*x*,*y*){*stuff*}**

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	
l	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:

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

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

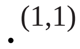
\pslabelsep

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).

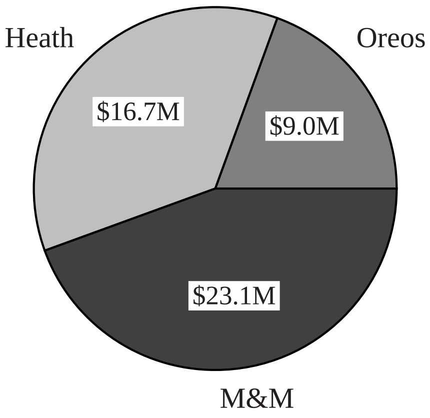
Here is a simple example:



```
\qdisk(1,1){1pt}
\uput[45](1,1){(1,1)}
```

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

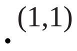
```
\psset{unit=1.2cm}
\pspicture(-2.2,-2.2)(2.2,2.2)
   \pswedge[fillstyle=solid,fillcolor=gray]{2}{0}{70}
   \pswedge[fillstyle=solid,fillcolor=lightgray]{2}{70}{200}
   \pswedge[fillstyle=solid,fillcolor=darkgray]{2}{200}{360}
   \SpecialCoor
   \psset{framesep=1.5pt}
   \rput(1.2;35){\psframebox*{\small\$9.0M}}
   \uput{2.2}[45](0,0){Oreos}
   \rput(1.2;135){\psframebox*{\small\$16.7M}}
   \uput{2.2}[135](0,0){Heath}
   \rput(1.2;280){\psframebox*{\small\$23.1M}}
   \uput{2.2}[280](0,0){M\&M}
\endpspicture
```



You can use the following abbreviations for *refangle*, which indicate the direction the angle points:[12][13]

Letter	*Short for*	*Equiv. to*	*Letter*	*Short for*	*Equiv. to*
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:



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

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

25 Repetition

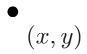
The macro

---

[12] Using the abbreviations when applicable is more efficient.

[13] There 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 **\rput**'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){$(x,y)$}
```



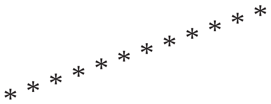
Here is the equivalence between **\uput**'s *refangle* abbreviations and **\Rput**'s *refpoint* abbreviations:

\uput	r	u	l	d	ur	ul	dr	dl
\Rput	l	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 (*x0*,*y0*) and advancing by (*x1*,*y1*) each time. (*x0*,*y0*) and (*x1*,*y1*) are always interpreted as Cartesian coordinates. For example:



```
\multirput(.5,0)(.3,.1){12}{*}
```

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

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

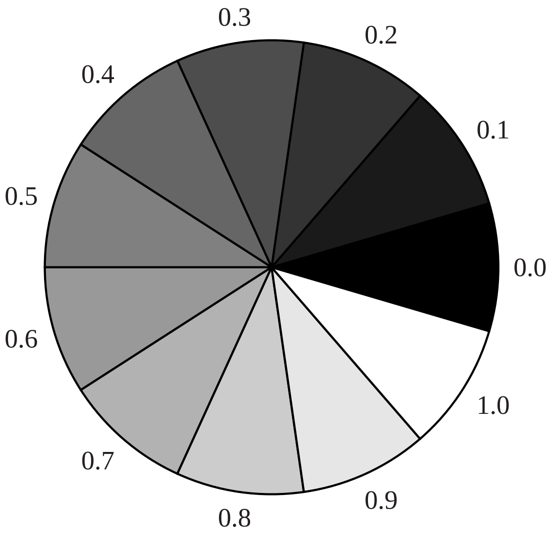
graphics can be one or more of the pure graphics objects described in Part II, or **\pscustom**. Note that **\multips** has the same syntax as **\multirput**, except that there is no *refpoint* argument (since the graphics are zero dimensional anyway). Also, unlike **\multirput**, the coordinates can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

```
\def\zigzag{\psline(0,0)(.5,1)(1.5,-1)(2,0)}%
\psset{unit=.25,linewidth=1.5pt}
\multips(0,0)(2,0){8}{\zigzag}
```

multido

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:

```
\begin{pspicture}(-3.4,-3.4)(3.4,3.4)
  \definecolor{mygray}{gray}{0}% Initialize 'mygray' for benefit
  \psset{fillstyle=solid,fillcolor=mygray}   % of this line.
  \SpecialCoor
  \degrees[1.1]
  \multido{\n=0.0+0.1}{11}{%
    \definecolor{mygray}{gray}{\n}%
    \psset{fillcolor=mygray}%
    \rput{\n}{\pswedge{3}{-.05}{.05}}
    \uput{3.2}[\n](0,0){\small\n}}
\end{pspicture}
```



All of these loop macros can be nested.

26 Axes

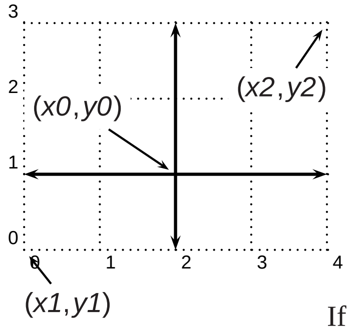
pst-plot

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:

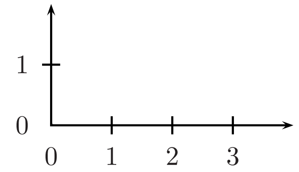
\psaxes*[*par*]{*arrows*}(*x0*,*y0*)(*x1*,*y1*)**(*x2*,*y2*)**

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:



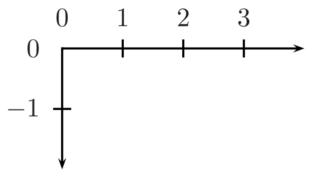
```
\psaxes[linewidth=1.2pt,labels=none,
  ticks=none]{<->}(2,1)(0,0)(4,3)
```

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)
```

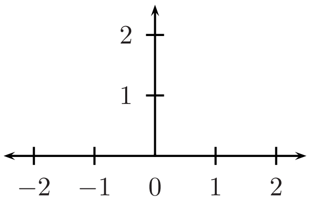
Labels (numbers) are put next to the axes, on the same side as *x1* and *y1*. Thus, if we enclose a different orthont, the numbers end up in the right place:



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

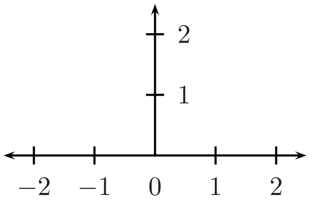
Also, if you set the **arrows** parameter, the first arrow is used for the tips at *x1* and *y1*, while the second arrow is used for the tips at *x2* and *y2*. Thus, in the preceding examples, the arrowheads ended up in the right place too.[14]

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:

---

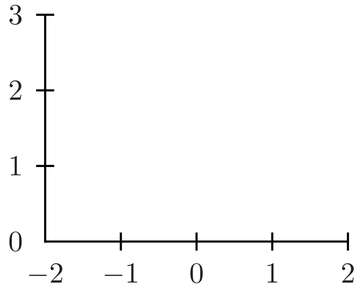
[14] Including a first arrow in these examples would have had no effect because arrows are never drawn at the origin.

Horitontal	*Vertical*	*Dflt*	*Description*
Ox=num	**Oy=num**	0	Label at origin.
Dx=num	**Dy=num**	1	Label increment.
dx=dim	**dy=dim**	0pt	Dist btwn labels.

When **dx** is 0, Dx\psxunit is used instead, and similarly for **dy**. Hence, the default values of 0pt 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:



```
\psaxes[Ox=-2](-2,0)(2,3)
```

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.

---

[15] For 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.

ticksize=*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 (ome 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

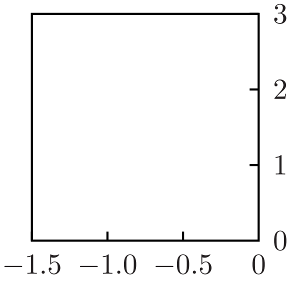
```
\def\pshlabel#1{\small  #1}
```

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 paremeters apply.

For example:



```
\psaxes[Dx=.5,dx=1,tickstyle=top,axesstyle=frame](-3,3)
```

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 **\multips** 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 LaTeX'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.

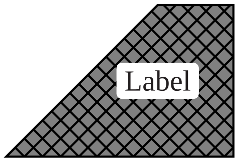
boxsep=*true/false* **Default: true**

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 TeX. 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,



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

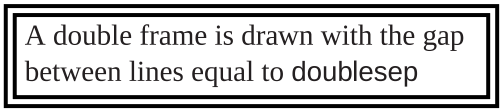
\psdblframebox*[*par*]**{*stuff*}**

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

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

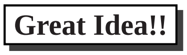
For example,

```
\psdblframebox[linewidth=1.5pt]{%
  \parbox[c]{6cm}{\raggedright A double frame is drawn
  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.

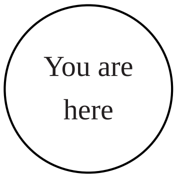


```
\psshadowbox{\textbf{Great Idea!!}}
```

You can get the shadow with **\psframebox** just be setting the **shadowsize** parameter, but with **\psframebox** 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:



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

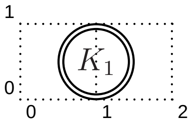
is distributed with

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

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

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

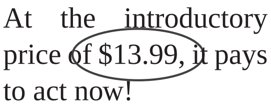
but it is more efficient. Unlike the **\rput** command, there is no argument for changing the reference point; it is always the center of the box. Instead, there is an optional argument for changing graphics parameters. For example



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

\psovalbox*[*par*]**{*stuff*}**

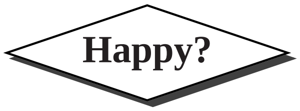
This draws an ellipse. If you want an oval with square sides and rounded corners, then use **\psframebox** with a positive value for **rectarc** or **linearc** (depending on whether **cornersize** is relative or absolute). Here is an example that uses **boxsep=false**:



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

\psdiabox*[*par*]**{*stuff*}**

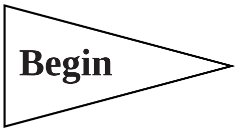
\psdiabox draws a diamond.



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

\pstribox*[*par*]**{*stuff*}**

\pstribox draws a triangle.

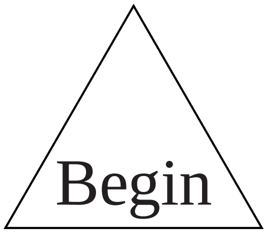


```
\pstribox[trimode=R,framesep=5pt]{\Large\textbf{Begin}}
```

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.



```
\pstribox[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 LaTeX's \makebox command.

28 Clipping

The command

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

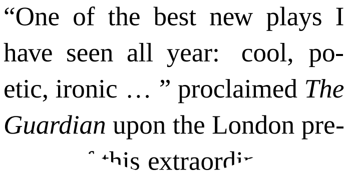
puts *stuff* in an \hbox and then clips around the boundary of the box, at a distance *dim* from the box (the default is 0pt).

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

The command[16]:

\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 pre-
this extraordi

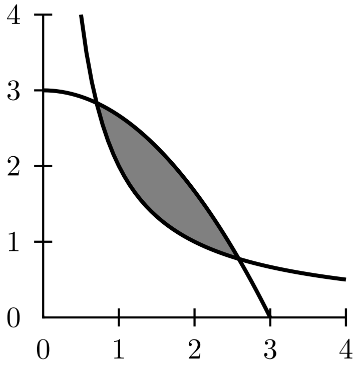
```
\parbox{4.5cm}{%
 \psclip{\psccurve[linestyle=none](-3,-2)(0.3,-1.5)(2.3,-2)
     (4.3,-1.5)(6.3,-2)(8,-1.5)(8,2)(-3,2)}
 “One of the best new plays I have seen all year: cool,
 poetic, ironic \ldots” proclaimed \emph{The Guardian} upon
 the London premiere of this extraordinary play about a Czech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

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:

---

[16] LATEX users can write \begin{pspclip} … \end{psclip} instead.

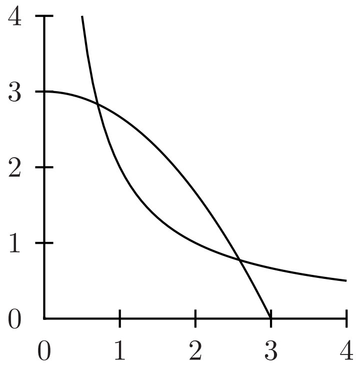


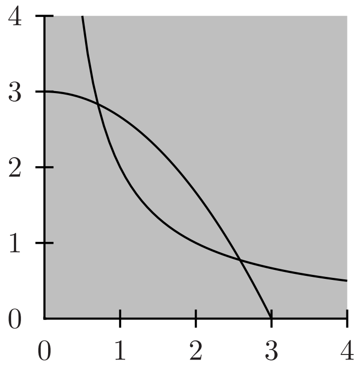
```
\psclip{%
  \pscustom[linestyle=none]{%
    \psplot{.5}{4}{2 x div}
    \lineto(4,4)}
  \pscustom[linestyle=none]{%
    \psplot{0}{3}{3 x x mul 3 div sub}
    \lineto(0,0)}}
\psframe*[linecolor=gray](0,0)(4,4)
\endpsclip
\psplot[linewidth=1.5pt]{.5}{4}{2 x div}
\psplot[linewidth=1.5pt]{0}{3}{3 x x mul 3 div sub}
\psaxes(4,4)
```

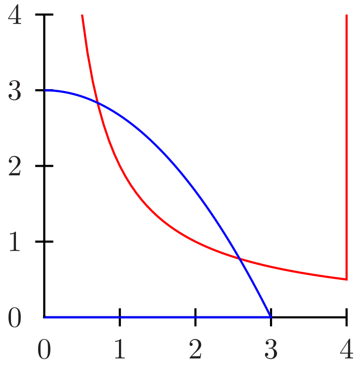
This is important to clearly understand this mechanism, so we give here the various steps of how it works in the preceding example:

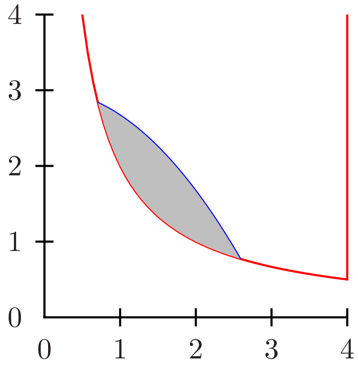
```
\pspicture(-0.5,-1)(4,4)
  \psaxes(4,4)
  \psplot{0.5}{4}{2 x div}
  \psplot{0}{3}{3 x x mul 3 div sub}
\endpspicture
\hfill
\pspicture(-0.5,-1)(4,4)
  \psframe*[linecolor=lightgray](4,4)
  \psaxes(4,4)
  \psplot{0}{3}{3 x x mul 3 div sub}
  \psplot{0.5}{4}{2 x div}
\endpspicture

\pspicture(-0.5,-1)(4,4)
  \psaxes(4,4)
  \pscustom[linecolor=red]{%
    \psplot{0.5}{4}{2 x div}
    \lineto(4,4)}
  \pscustom[linecolor=blue]{%
    \psplot{0}{3}{3 x x mul 3 div sub}
    \lineto(0,0)}
\endpspicture
\hfill
\pspicture(-0.5,-1)(4,4)
  \psaxes(4,4)
  \psclip{\pscustom[linecolor=red]{%
            \psplot{0.5}{4}{2 x div}
            \lineto(4,4)}
          \pscustom[linecolor=blue]{%
            \psplot{0}{3}{3 x x mul 3 div sub}
            \lineto(0,0)}}
    \psframe*[linecolor=lightgray](4,4)
  \endpsclip
\endpspicture

\pspicture(-0.5,-1)(4,4)
```

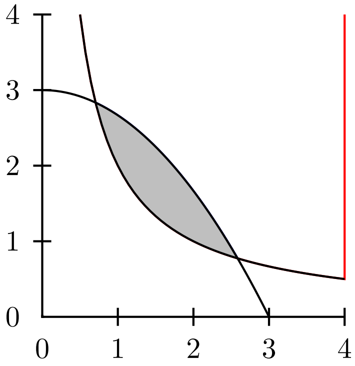
```
    \psaxes(4,4)
    \psclip{\pscustom[linecolor=red]{%
                \psplot{0.5}{4}{2 x div}
                \lineto(4,4)}
            \pscustom[linecolor=blue]{%
                \psplot{0}{3}{3 x x mul 3 div sub}
                \lineto(0,0)}}
        \psframe*[linecolor=lightgray](4,4)
    \endpsclip
    \psplot{0}{3}{3 x x mul 3 div sub}
    \psplot{0.5}{4}{2 x div}
\endpspicture
\hfill
\pspicture(-0.5,-1)(4,4)
    \psaxes(4,4)
    \psclip{\pscustom[linestyle=none]{%
                \psplot{0.5}{4}{2 x div}
                \lineto(4,4)}
            \pscustom[linestyle=none]{%
                \psplot{0}{3}{3 x x mul 3 div sub}
                \lineto(0,0)}}
        \psframe*[linecolor=lightgray](4,4)
    \endpsclip
    \psplot{0}{3}{3 x x mul 3 div sub}
    \psplot{0.5}{4}{2 x div}
\endpspicture
```

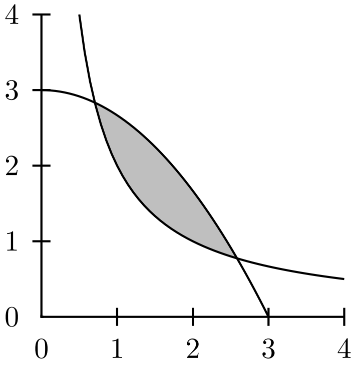












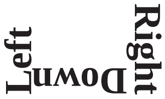
Driver notes: The clipping macros use **\pstverbscale** and **\pstVerb**. Don't be surprised if PSTricks's clipping does not work or causes problem—it is never robust. \endpsclip uses initclip. This can interfere with other clipping operations, and especially if the TEX document is converted to an Encapsulated 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:



```
\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 ƨidɈ (\scalebox{-1 1}{this}).

\scaleboxto(*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

```
\scaleboxto(4,2){Big  and  long}
```

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

```
\pslongbox{Rotateleft}{\rotateleft}
\pslongbox{Rotateright}{\rotateright}
\pslongbox{Rotatedown}{\rotatedown}
\pslongbox{Scalebox}{\scalebox}
\pslongbox{Scaleboxto}{\scaleboxto}
```

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

Question: How do Democrats organize a firing squad?

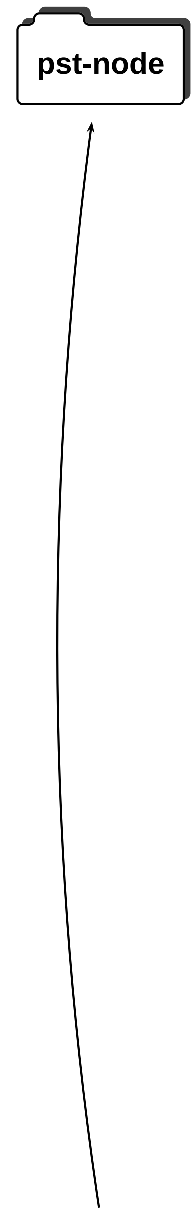
Answer: First they get in a circle, …

```
Question:  How  do  Democrats  organize  a  firing  squad?

\begin{Rotatedown}
  \parbox{\hsize}{Answer:  First  they  get  in  a  circle,  \ldots\hss}%
\end{Rotatedown}
```

See the documentation of the ‘fancybox’ package for tips on rotating a LaTeX 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.

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

30 Nodes

Nodes have a name. a boundary and a center.

> *Warning:* The name is for refering 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 visable object they create.

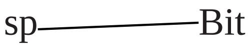
Here are the nodes:

\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**.

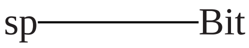
\Rnode*[*par*]***{name}{stuff}***

\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:



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

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



```
\Large
\Rnode{A}{sp} \hskip 2cm \Rnode{B}{Bit}
\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* **Default: 0**
vref=*dim* **Default: .7ex**

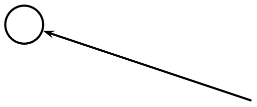
parameters. In the horizontal direction, the center is located fraction **href** from the center to the edge. E.g, if **href=-1**, the center is on the left edge of the box. In the vertical direction, the center is located distance **vref** from the baseline. The **vref** parameter is evaluated each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



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

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

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

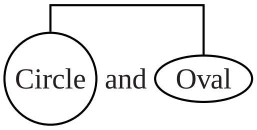
\cnodeput*[*par*]{*angle*}(*x*,*y*)***{name}{stuff}***

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

```
\rput{angle}(x,y){\circlenode{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 **\circlenode** and **\ovalnode**:



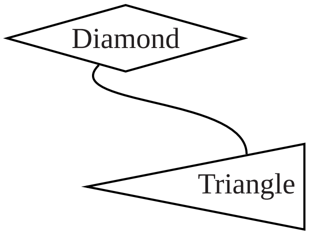
```
\circlenode{A}{Circle}  and  \ovalnode{B}{Oval}
\ncbar[angle=90]{A}{B}
```

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

This is like **\diabox**.

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

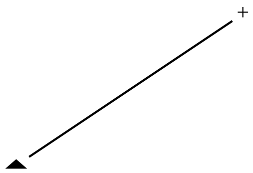
This is like **\tribox**.



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

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

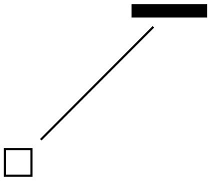
This is a variant of **\psdot**. For example:



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

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

The f stands for "frame". This is like, but easier than, putting a **\psframe** in an **\rnode**.



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

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

---

[17] The 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: 0pt**

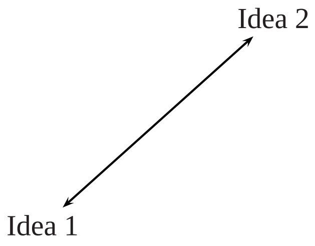
nodesep is the border around the nodes that is added for the purpose of determining where to connect the lines.

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*}**

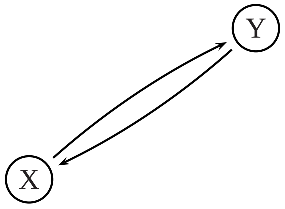
This draws a straight line between the nodes. For example:



```
\rput[bl](0,0){\rnode{A}{Idea 1}}
\rput[tr](4,3){\rnode{B}{Idea 2}}
\ncline[nodesep=3pt]{<->}{A}{B}
```

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

This connects the two nodes with an arc.



```
\cnodeput(0,0){A}{X}
\cnodeput(3,2){B}{Y}
\psset{nodesep=3pt}
\ncarc{->}{A}{B}
\ncarc{->}{B}{A}
```

---

[19] When 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, TEX will not report any errors.

The angle between the arc and the line between the two nodes is[20]

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**

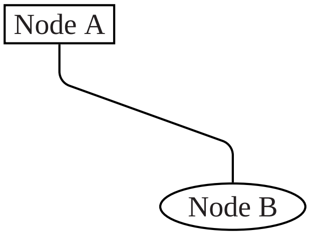
(and **armA** and **armB**) parameter.

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:

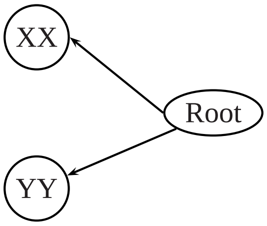


```
\rput[tl](0,3){\rnode{A}{\psframebox{Node A}}}
\rput[br](4,0){\ovalnode{B}{Node B}}
\ncdiag[angleA=-90, angleB=90, arm=.5, linearc=.2]{A}{B}
```

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:

---

[20] Rather 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 **??**.

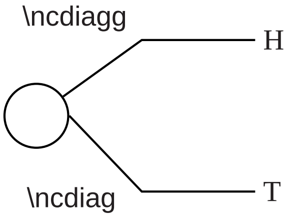


```
\rput[r](4,1){\ovalnode{R}{Root}}
\cnodeput(1,2){A}{XX}
\cnodeput(1,0){B}{YY}
\ncdiag[angleB=180,  arm=0]{<-}{A}{R}
\ncline{<-}{B}{R}
```

(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**:

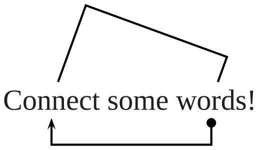


```
\cnode(0,0){12pt}{a}
\rput[l](3,1){\rnode{b}{H}}
\rput[l](3,-1){\rnode{c}{T}}
\ncdiagg[angleA=180,  armA=1.5,  nodesepA=3pt]{b}{a}
\ncdiag[angleA=180,  armA=1.5,  armB=0,  nodesepA=3pt]{c}{a}
```

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.

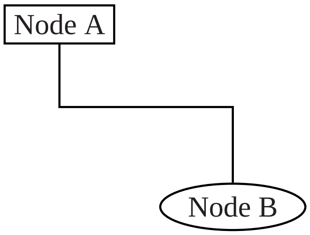


```
\rnode{A}{Connect}  some  \rnode{B}{words}!
\ncbar[nodesep=3pt,angle=-90]{<-**}{A}{B}
\ncbar[nodesep=3pt,angle=70]{A}{B}
```

Generally, the whole line has three straight segments.

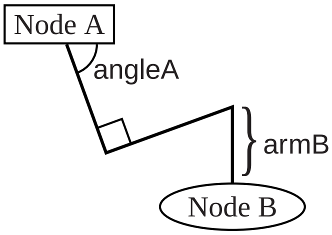
\ncangle*[*par*]{*arrows*}***{*nodeA*}{*nodeB*}**

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:



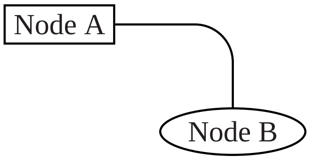
```
\rput[tl](0,3){\rnode{A}{\psframebox{Node  A}}}
\rput[br](4,0){\ovalnode{B}{Node  B}}
\ncangle[angleA=-90,angleB=90,armB=1cm]{A}{B}
```

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



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

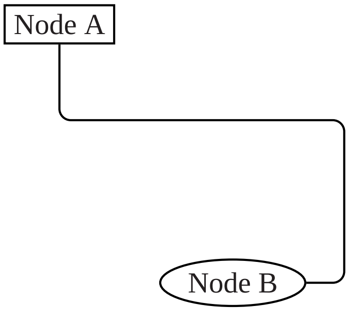
\ncangle is also a good way to join nodes by a right angle, with just two line segments, as in this example:



```
\rput[tl](0,2){\rnode{A}{\psframebox{Node  A}}}
\rput[br](4,0){\ovalnode{B}{Node  B}}
\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 he **armA** parameter. Arm A is connected to arm B by two line segments that eet 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:

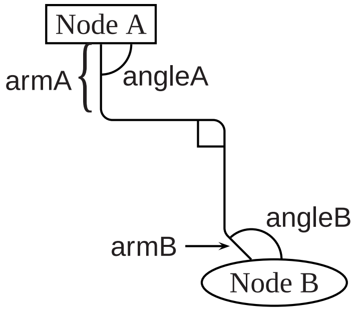


```
\rput[tl](0,4){\rnode{A}{\psframebox{Node  A}}}
\rput[br](4,0){\ovalnode{B}{Node  B}}
\ncangles[angleA=-90,  armA=1cm,  armB=.5cm,  linearc=.15]{A}{B}
```

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

---

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



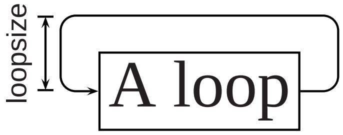
```
\rput[tl](0,4){\rnode{A}{\psframebox{Node  A}}}
\rput[br](4,0){\ovalnode{B}{Node  B}}
\ncangles[angleA=-90,  angleB=135,  armA=1cm,  armB=.5cm,
    linearc=.15]{A}{B}
```

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

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

loopsize=*dim* **Default: 1cm**

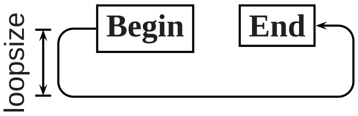
This segment connects to arm B the way arm A connects to arm B with **\ncline**; that is, two more segments are drawn, which join the first segment and each other at right angles, and then join arm B. For example:



```
\rnode{a}{\psframebox{\Huge  A  loop}}
\ncloop[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 **\ncloop** connects two different nodes:

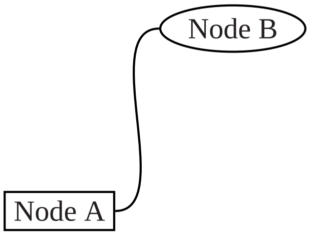


```
\parbox{3cm}{%
\rnode{A}{\psframebox{\large\textbf{Begin}}}
\vspace{1cm}\hspace*{\fill}
\rnode{B}{\psframebox{\large\textbf{End}}}
\ncloop[angleA=180,loopsize=.9,arm=.5,linearc=.2]{->}{A}{B}}
```

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.



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

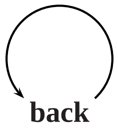
You specify the angle at which the curve joins the nodes by setting the **angle** (and **angleA** and **angleB**) parameter. The distance to the control points is set with the

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**.



```
\rnode{A}{\textbf{back}}
\nccircle[nodesep=3pt]{->}{A}{.7cm}
\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.

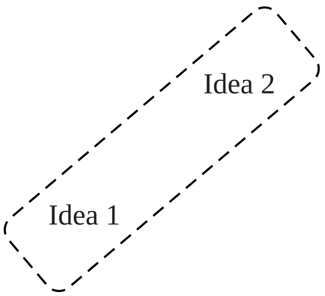
The last two node connections are also special. Rather than connecting the nodes with an open curve, they enclose the nodes in a box or curved box. You can think of them as variants of **\ncline** and **\ncarc**. In both cases, the half the width of the box is

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 **nodesepA** and **nodesepB**.

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

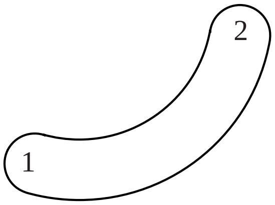
\ncbox encloses the nodes in a box with straight sides. For example:



```
\rput[bl](.5,0){\rnode{A}{Idea  1}}
\rput[tr](3.5,2){\rnode{B}{Idea  2}}
\ncbox[nodesep=.5cm,boxsize=.6,linearc=.2,
   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.



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

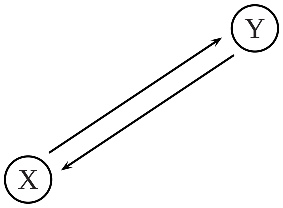
The arc is drawn counterclockwise from node A to node B.

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

offset=*dim* **Default: 0pt**

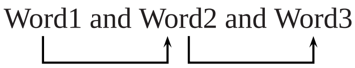
(You can also set **offsetA** and **offsetB** independently.) This shifts the point where the connection joins up by *dim* (given the convention that connections go from left to right).

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



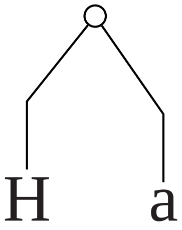
```
\cnodeput(0,0){A}{X}
\cnodeput(3,2){B}{Y}
\psset{nodesep=3pt,offset=4pt,arrows=->}
\ncline{A}{B}
\ncline{B}{A}
```

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:



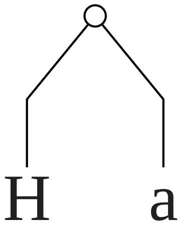
```
\rnode{A}{Word1} and \rnode{B}{Word2} and \rnode{C}{Word3}
\ncbar[offsetB=4pt,angleA=-90,nodesep=3pt]{->}{A}{B}
\ncbar[offsetA=4pt,angleA=-90,nodesep=3pt]{->}{B}{C}
```

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:



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



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

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

gives the same result as

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

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

Command	*Corresponds to:*
\pcline{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\ncline**
\pccurve{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\nccurve**
\pcarc{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\ncarc**
\pcbar{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\ncbar**
\pcdiag{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\ncdiag**
\pcdiagg{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\ncdiagg**
\pcangle{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\ncangle**
\pcangles{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\ncangles**
\pcloop{*arrows*}**(*x1*,*y1*)(*x2*,*y2*)**	**\ncloop**
\pcbox(*x1*,*y1*)(*x2*,*y2*)	**\ncbox**
\pcarcbox(*x1*,*y1*)(*x2*,*y2*)	**\ncarcbox**

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 TEX 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:

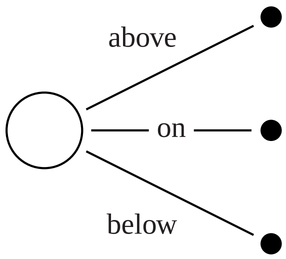
\ncput*[*par*]**{*stuff*}**
\naput*[*par*]**{*stuff*}**
\nbput*[*par*]**{*stuff*}**

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

\ncput	*on* the line
\naput	*above* the line
\nbput	*below* the line

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

Here is an example:



```
\cnode(0,0){.5cm}{root}
\cnode*(3,1.5){4pt}{A}
\cnode*(3,0){4pt}{B}
\cnode*(3,-1.5){4pt}{C}
\psset{nodesep=3pt}
\ncline{root}{A}
\naput{above}
\ncline{root}{B}
\ncput*{on}
\ncline{root}{C}
\nbput{below}
```

\naput and **\nbput** use the same algorithm as **\uput** for displacing the labels, and the distance beteen 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:

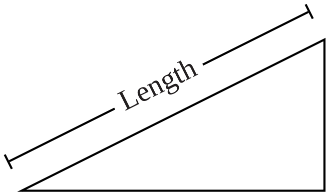
nrot=*rot* **Default: 0**

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

```
{:angle}
```

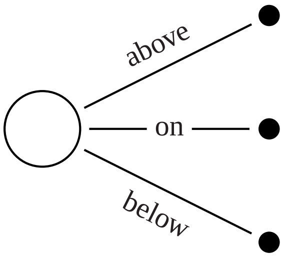
The angle is then measured with respect to the node connection. E.g., if the angle is {:U}, then the label runs parallel to the node connection. Since the label can include other put commands, you really have a lot of control over the label position.

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



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

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



```
\cnode(0,0){.5cm}{root}
\cnode*(3,1.5){4pt}{A}
\cnode*(3,0){4pt}{B}
\cnode*(3,-1.5){4pt}{C}
\psset{nodesep=3pt,nrot=:U}
\ncline{root}{A}
\naput{above}
\ncline{root}{B}
\ncput*{on}
\ncline{root}{C}
\nbput{below}
```

The position on the node connection is set by the

npos=*num* **Default:**

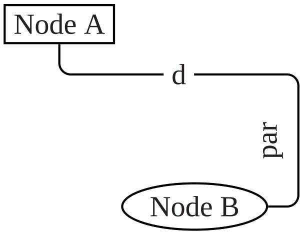
parameter, roughly according to the following scheme: Each node connection has potentially one or more segments, including the arms and connecting lines. A number **npos** between 0 and 1 picks a point on the first segment from node A to B (fraction **npos** from the beginning to the end of the segment),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

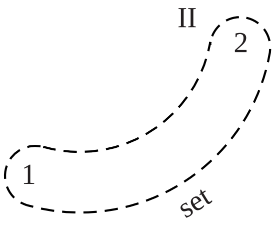
Connection	*Segments*	*Range*	*Default*
\ncline	1	$0\leq pos\leq 1$	0.5
\nccurve	1	$0\leq pos\leq 1$	0.5
\ncarc	1	$0\leq pos\leq 1$	0.5
\ncbar	3	$0\leq pos\leq 3$	1.5
\ncdiag	3	$0\leq pos\leq 3$	1.5
\ncdiagg	2	$0\leq pos\leq 2$	0.5
\ncangle	3	$0\leq pos\leq 3$	1.5
\ncangles	4	$0\leq pos\leq 4$	1.5
\ncloop	5	$0\leq pos\leq 5$	2.5
\nccircle	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:



```
\rput[tl](0,3){\rnode{A}{\psframebox{Node  A}}}
\rput[br](3.5,0){\ovalnode{B}{Node  B}}
\ncangles[angleA=-90,arm=.4cm,linearc=.15]{A}{B}
\ncput*{d}
\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.

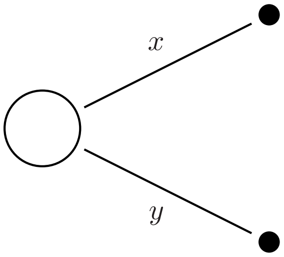


```
\rput[bl](.5,0){\rnode{A}{1}}
\rput[tr](3.5,2){\rnode{B}{2}}
\ncarcbox[nodesep=.2cm,boxsize=.4,linearc=.4,
   arcangle=50,linestyle=dashed]{<->}{A}{B}
\nbput[nrot=:U]{set}
\nbput[npos=2]{II}
```

If you set the parameter

shortput=*none/nab/tablr/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**.



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

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]

---

[22] You 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=tablr** and **shortput=tab** options are described on pages 75 and **??**, respectively.

33 Node connection labels: II

Now the second group of node connections:

\tvput*[*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.

\tvput, **\tlput** and **\trput** find the position that lies fraction *tpos* in the *vertical* direction from the upper node to the lower node. **\thput**, **\taput** and **\tbput** 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*
\tvput	vertical	middle
\tlput	vertical	left
\trput	vertical	right
\thput	horizontal	middle
\taput	horizontal	above
\tbput	horizontal	below

Here is an example:

```
\[
  \setlength{\arraycolsep}{1.1cm}
  \begin{array}{cc}
    \Rnode{a}{(X-A)}  &  \Rnode{b}{A}  \\[1.5cm]
    \Rnode{c}{x} & \Rnode{d}{\tilde{X}}
  \end{array}
  \psset{nodesep=5pt,arrows=->}
  \everypsbox{\scriptstyle}
  \ncline{a}{c}\tlput{r}
  \ncline{a}{b}\taput{u}
  \ncline[linestyle=dashed]{c}{d}\tbput{b}
  \ncline{b}{d}\trput{s}
\]
```

$$\begin{tikzcd}
(X-A) \arrow[r, "u"] \arrow[d, "r"'] & A \arrow[d, "s"] \\
x \arrow[r, dashed, "b"'] & \tilde{X}
\end{tikzcd}$$

$$\begin{tikzcd}
(X-A) \arrow[r, "u"] \arrow[d, "r"'] & a \arrow[d, "s"] \\
x \arrow[r, dashed, "b"'] & \tilde{X}
\end{tikzcd}$$

On the left is the diagram with **\tlput**, **\trput** **\tbput** and **\Rnode**, as shown in the code. On the right is the same diagram, but with **\naput** **\nbput** and **\rnode**.

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=tablr**, then you can use the following single-character abbreviations for the t put commands:

Char.	*Short for:*
^	**\taput**
_	**\tbput**
<	**\tlput**
>	**\trput**

You can change the character abbreviations with

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

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

Driver notes: The node macros use **\pstVerb** and **\pstverbscale**.

34 Attaching labels to nodes

The command

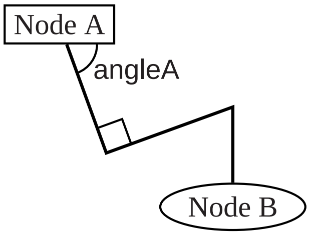
\nput*[*par*]**{*refangle*}{*name*}{*stuff*}**

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 **\nput** to mark an angle in a previous example:



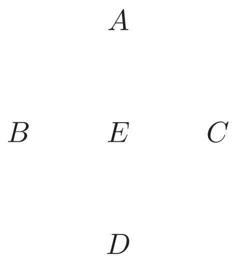
```
\rput[br](4,0){\ovalnode{B}{Node  B}}
\rput[tl](0,3){\rnode{A}{\psframebox{Node  A}}}
\nput[labelsep=0]{-70}{A}{%
   \psarcn(0,0){.4cm}{0}{-70}
   \uput{.4cm}[-35](0,0){\texttt{angleA}}}
\ncangle[angleA=-70,angleB=90,armB=1cm,linewidth=1.2pt]{A}{B}
\ncput[nrot=:U,npos=1]{\psframe[dimen=middle](0,0)(.35,.35)}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

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 \halignprimitive, LaTeX's tabular environment, and AMS-TeX's \matrix, but PSTricks contains its own alignment environment that is especially adapted for this purpose[24]

\psmatrix{} ... \endpsmatrix

Here is an example



```
$
\psmatrix[colsep=1cm,rowsep=1cm]
     &  A  \\
  B  &  E  &  C  \\
     &  D  &
\endpsmatrix
$
```

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

---

[23] Not to be confused with the nput parameter.

[24] LaTeX users can instead write:

\begin{psmatrix} *stuff* \end{psmatrix}

as many columns as you need. The entries are horizontally centered. Rows are ended by \\. **\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:

$$\begin{tikzcd}
 & X \arrow[dl, "f"'] \arrow[d, "g"] \\
Y \arrow[r, dotted, "h"'] & Z
\end{tikzcd}$$

```
$
\psmatrix[colsep=1cm]
      & X \\
   Y & Z
\endpsmatrix
\everypsbox{\scriptstyle}%
\psset{nodesep=3pt,arrows=->}
\ncline{1,2}{2,1}
\tlput{f}
\ncline{1,2}{2,2}
\trput{g}
\ncline[linestyle=dotted]{2,1}{2,2}
\tbput{h}
$
```

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

$$\begin{tikzcd}
U \arrow[dr, "y"'] \arrow[drr, equal, dashed, "x"] & & \\
 & X \times_Z Y \arrow[r, "p"'] \arrow[d, "q"'] & X \arrow[d, "f"] \\
 & Y \arrow[r, "g"'] & Z
\end{tikzcd}$$

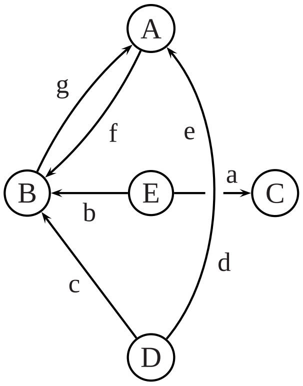
```
$
   \begin{psmatrix}
      U \\
         &   X\times_Z Y  &  X \\
         &         Y          &  Z
      \psset{arrows=->,nodesep=3pt}
      \everypsbox{\scriptstyle}
      \ncline{1,1}{2,2}_{y}
      \ncline[doubleline=true,linestyle=dashed]{-}{1,1}{2,3}^{x}
      \ncline{2,2}{3,2}<{q}
      \ncline{2,2}{2,3}_{p}
      \ncline{2,3}{3,3}>{f}
      \ncline{3,2}{3,3}_{g}
   \end{psmatrix}
$
```

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:



```
\psmatrix[mnode=circle,colsep=1]
      & A \\
   B & E & C \\
      & D &
\endpsmatrix
\psset{shortput=nab,arrows=->,labelsep=3pt}
\small
\ncline{2,2}{2,3}^[npos=.75]{a}
\ncline{2,2}{2,1}^{b}
\ncline{3,2}{2,1}^{c}
\ncarc[arcangle=-40,border=3pt]{3,2}{1,2}
   _[npos=.3]{d}^[npos=.7]{e}
\ncarc[arcangle=12]{1,2}{2,1}^{f}
\ncarc[arcangle=12]{2,1}{1,2}^{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:

$$\begin{tikzcd}
 & \boxed{X} \arrow[dl] \arrow[d] \\
Y \arrow[r, dotted] & Z
\end{tikzcd}$$

```
$
\psmatrix[colsep=1cm]
      & [mnode=circle] X \\
   Y & Z
\endpsmatrix
\psset{nodesep=3pt,arrows=->}
\ncline{1,2}{2,1}
\ncline{1,2}{2,2}
\ncline[linestyle=dotted]{2,1}{2,2}
$
```

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

  nodealign=*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 LaTeX, 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 athe 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 TeX’s **\multispan**, or LaTeX’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]):

```
$
\psmatrix[rowsep=1,colsep=1]
  % Nodes
  % First line
                  & [name=SL] \Sigma^L &
      & & & [name=SR] \Sigma^R\\[0cm]
  % Second line
  [name=L]  L    &                        & [name=Lr]  L_r
      & & [name=R] R\\[1.5cm]
  % Third line
  [name=Lm] L_m &                         & [name=Krm] K_{r,m}
      & & [name=Rm] R_{m^*}\\
                  & [name=SG] \Sigma^G &
      & & & [name=SH] \Sigma^H\\[0cm]
  % Fourth line
  [name=G]  G    &                         & [name=Gr]  G_{r^*}
```

```
        &  &  [name=H]  H
    %
    %  Links
    \everypsbox{\scriptstyle}
    \psset{arrows=->,nodesep=2mm,border=3pt}%
    \ncline{Lr}{R}        ^{r}
    \ncline{Krm}{Rm}  ^{r}
    \ncline{Gr}{H}        _{r^*}
    \ncline{Lr}{L}        ^[tpos=0.3]{i_1}
    \ncline{Krm}{Lm}  ^[tpos=0.3]{i_3}
    \ncline{Gr}{G}        _{i_5}
    \ncline{SL}{SR}    ^{\varphi^r}
    \ncline{SG}{SH}    _{\varphi^{r^*}}
    \ncline{SR}{SH}    >{\varphi^{m^*}}
    \ncline{SL}{SG}    <{\varphi^m}
    \ncline{Lm}{G}      <{m}
    \ncline{Krm}{Lr}    >{i_4}
    \ncline{Rm}{H}      >[tpos=0.3]{m^*}
    \ncline{Lm}{L}        <{i_2}
    \ncline{Krm}{Gr}    >[tpos=0.3]{m}
    \ncline{Rm}{R}      >{i_6}
    \ncline{L}{SL}        <[tpos=0.3]{\lambda^L}
    \ncline{R}{SR}      >[tpos=0.6]{\lambda^R}
    \ncline{G}{SG}      <[tpos=0.3]{\lambda^G}
    \ncline[linestyle=dashed]{H}{SH}  >[tpos=0.6]{\lambda^H}
\endpsmatrix
$
```

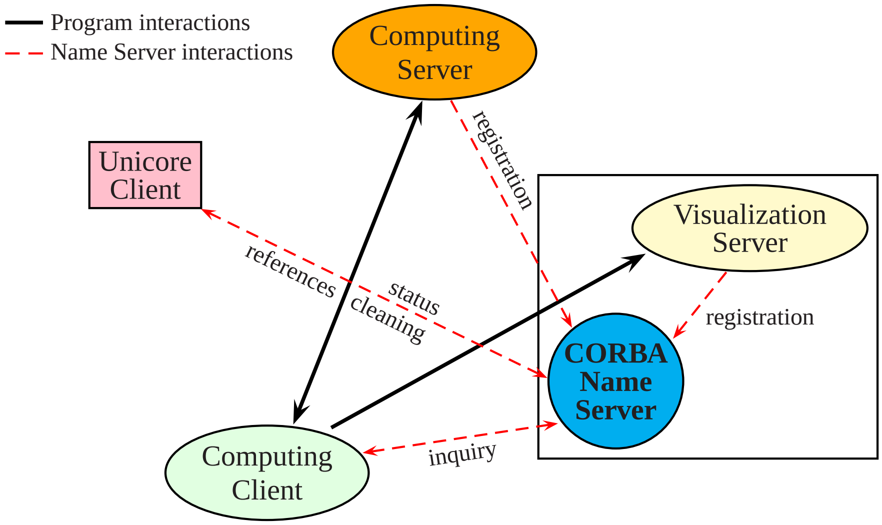
$$\begin{tikzcd}
 & \Sigma^L \arrow[rrr, "\varphi^r"] \arrow[ddd, "\varphi^m"'] & & & \Sigma^R \arrow[ddd, "\varphi^{m^*}"] \\
L \arrow[ur, "\lambda^L"] & & L_r \arrow[ll, "i_1"'] \arrow[r, "r"] & R \arrow[ur, "\lambda^R"'] & \\
L_m \arrow[u, "i_2"] \arrow[dd, "m"'] & & K_{r,m} \arrow[ll, "i_3"'] \arrow[r, "r"] \arrow[u, "i_4"] \arrow[dd, "m"] & R_{m^*} \arrow[u, "i_6"] \arrow[dd, "m^*"] & \\
 & \Sigma^G \arrow[rrr, "\varphi^{r^*}"'] & & & \Sigma^H \\
G \arrow[ur, "\lambda^G"] & & G_{r^*} \arrow[ll, "i_5"'] \arrow[r, "r^*"'] & H \arrow[ur, dashed, "\lambda^H"'] &
\end{tikzcd}$$

And second, a general connection diagram:

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

```
  \shortstack{Computing\\Server}}}

\newcommand{\PstUnicoreClient}{%
\psframebox[fillstyle=solid,fillcolor=Pink]{%
  \shortstack{Unicore\\Client}}}

\newcommand{\PstVisualizationServer}{%
\psovalbox[fillstyle=solid,fillcolor=LemonChiffon]{%
  \shortstack{Visualization\\Server}}}

\newcommand{\PstCorbaNameServer}{%
\pscirclebox[fillstyle=solid,fillcolor=cyan,framesep=0.02]{%
  \bfseries\shortstack{CORBA\\Name\\Server}}}

\newcommand{\PstComputingClient}{%
\psovalbox[fillstyle=solid,fillcolor=PaleGreen]{%
  \shortstack{Computing\\Client}}}

\rput[l](-0.5,6){%
  \footnotesize
  \shortstack[l]{%
    \psline[linewidth=0.05,linecolor=black](-0.6,0.1)(-0.1,0.1)
    Program interactions\\
    \psline[linestyle=dashed,linecolor=red](-0.6,0.08)(-0.1,0.08)
    Name Server interactions}}
%
\begin{psmatrix}[colsep=-0.5,rowsep=-0.5]
    & & [name=ComputingServer] \PstComputingServer\\[1cm]
  [name=Unicore] \PstUnicoreClient\\
    & & & \psframebox{%
            \begin{psmatrix}[colsep=-0.7,rowsep=0]
             & [name=Visualization]\PstVisualizationServer\\[5mm]
             [name=CORBA] \PstCorbaNameServer
            \end{psmatrix}}\\
    & [name=ComputingClient] \PstComputingClient
\end{psmatrix}
%
\ncline[arrowscale=1.5,linewidth=0.05]{<->}
     {ComputingServer}{ComputingClient}
\ncline[arrowscale=1.5,linewidth=0.05,offsetA=0.35,offsetB=0.5,
      nodesepA=-0.35,nodesepB=0.2]{->}
     {ComputingClient}{Visualization}
\psset{linestyle=dashed,linecolor=red,arrowscale=1.3,labelsep=0.06}%
\ncline[offsetA=0.08,nodesepA=0.07]{<->}{Unicore}{CORBA}
  \naput[npos=0.6,nrot=:U]{\footnotesize status}
  \nbput[npos=0.41,nrot=:U]{\footnotesize references \ \ cleaning}
\ncline[nodesepB=-0.2]{->}{ComputingServer}{CORBA}
  \naput[npos=0.3,nrot=:U]{\footnotesize registration}
\ncline[nodesepB=-0.15]{->}{Visualization}{CORBA}
  \naput{\footnotesize registration}
\ncline[offsetA=0.2,nodesepA=-0.05,nodesepB=-0.14]{<->}
     {ComputingClient}{CORBA}
```

```
\nbput[nrot=:U]{\footnotesize  inquiry}
```



VIII Trees

37 Overview

pst-tree

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.

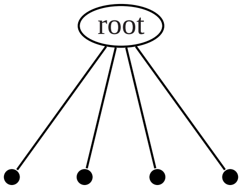
The tree commands are

\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:



```
\pstree[radius=3pt]{\Toval{root}}{\TC*  \TC*  \TC*  \TC*}
```

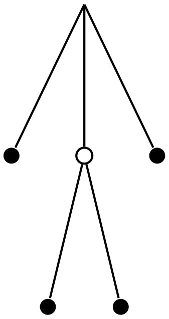
There is no difference between a terminal node and a root node, other than their position in the **\pstree** command.

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

---

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

[26] LaTeX users can write \begin{psTree} and \end{psTree} instead.



```
\pstree[radius=3pt]{\Tp}{%
  \TC*
  \pstree{\TC}{\TC*  \TC*}
  \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 `\Toval`. 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*}**
\Tdia*[*par*]**{*stuff*}**
\Ttri*[*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:

```
\small
\psset{armB=1cm, levelsep=3cm, treesep=-3mm,
  angleB=-90, angleA=90, nodesepA=3pt}
\def\s#1{#1~{\tt\string#1}}
\def\b#1{#1{\tt\string#1}}
\def\psedge#1#2{\ncangle{#2}{#1}}
\psTree[treenodesize=1cm]{\Toval{Tree nodes}}
  \s\Tp
  \Tc{.5}~{\tt\string\Tc}
  \s\TC
  \psTree[levelsep=4cm,armB=2cm]{\Tp[edge=\ncline]}
    \b\Tcircle
    \s\Tdot
    \TCircle[radius=1]{\tt\string\TCircle}
    \Tn
    \b\Toval
    \b\Ttri
    \b\Tdia
  \endpsTree
  \s\Tf
  \b\Tr
  \b\TR
\endpsTree
```

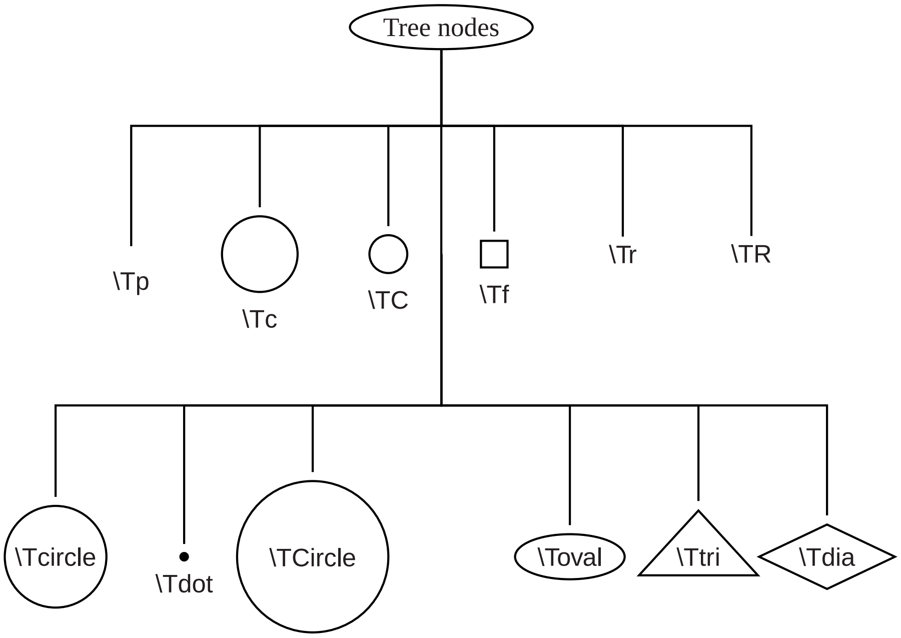
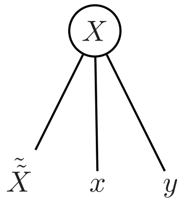
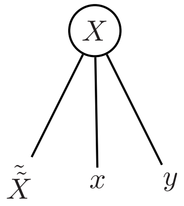


Figure 1: The tree nodes.



```
$
\pstree[nodesepB=3pt]{\Tcircle{X}}{%
   \TR{\tilde{\tilde{X}}}
   \TR{x}
   \TR{y}}
$
```

Compare with this example, which uses **\Tr**:



```
$
\pstree[nodesepB=3pt]{\Tcircle{X}}{%
   \Tr{\tilde{\tilde{X}}}
   \Tr{x}
   \Tr{y}}
$
```

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.

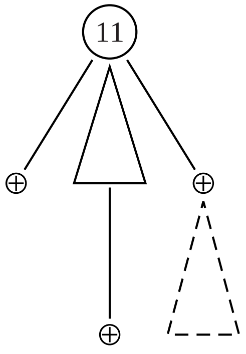
There is also a special tree node that doesn't have a "normal" version and that can't be used as the root node of a whole tree:

\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:



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

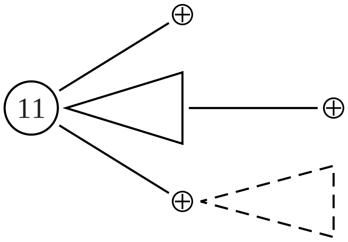
39 Tree orientation

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

treemode=*D/U/R/L* **Default: D**

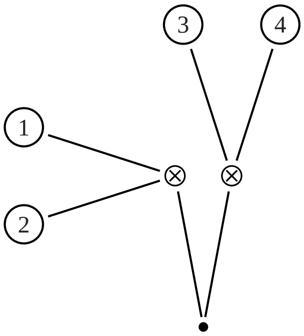
parameter.

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



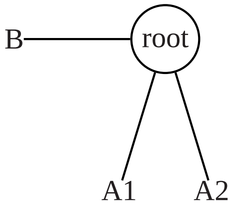
```
\pstree[dotstyle=oplus,dotsize=8pt,nodesep=2pt,treemode=R]
  {\Tcircle{11}}{%
    \Tdot
    \pstree{\Tfan}{\Tdot}
    \pstree{\Tdot}{\Tfan[linestyle=dashed]}}
```

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:



```
\footnotesize
\pstree[treemode=U,dotstyle=otimes,dotsize=8pt,nodesep=2pt]
  {\Tdot}{%
    \pstree[treemode=L]{\Tdot}{\Tcircle{1}  \Tcircle{2}}
    \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*.

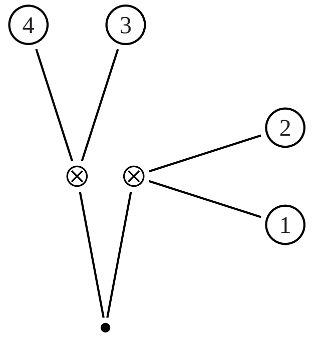


```
\pstree{\pstree[treemode=L]{\Tcircle{root}}{\Tr{B}}}{%
  \Tr{A1}
  \Tr{A2}}
```

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:



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

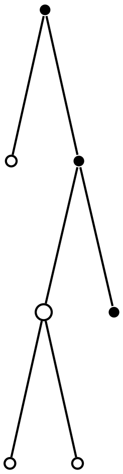
tight When **treefit=tight**, which is the default, **treesep** is the minimum distance between each of the levels of the subtrees.

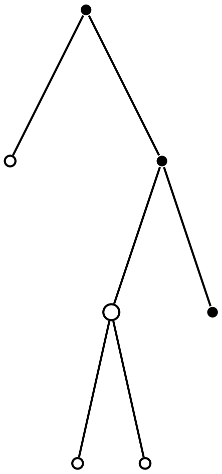
loose When **treefit=loose**, **treesep** is the distance between the subtrees' bounding boxes. Except when you have large intermediate nodes, the effect is that the horizontal distance (or vertical distance, for horizontal trees) between all the terminal nodes is the same (even when they are on different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

Compare:

---

[27] When 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 **treefit=loose**, trees take up more space, but sometimes the structure of the tree is emphasized.

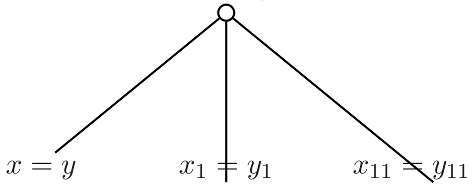
Sometimes you want the spacing between the centers of the nodes to be regular even though the nodes have different sizes. If you set

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*.

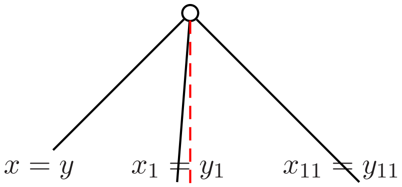
For example, ternary trees look nice when they are symmetric, as in the following example:

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



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

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

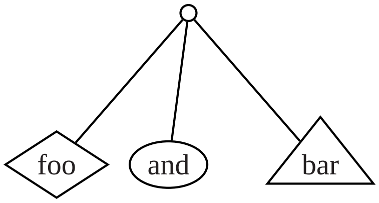


```
\pstree[nodesepB=-8pt]{\Tc{3pt}}{%
  \TR{$x=y$}
  \TR{$x_1=y_1$}
  \TR{$x_{11}=y_{11}$}}
```

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

\tspace{*dim*}

between them:



```
\pstree{\Tc{3pt}}{%
  \Tdia{foo}
  \tspace{-0.5}
  \Toval{and}
  \Ttri{bar}}
```

41 Spacing between the root and successors

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

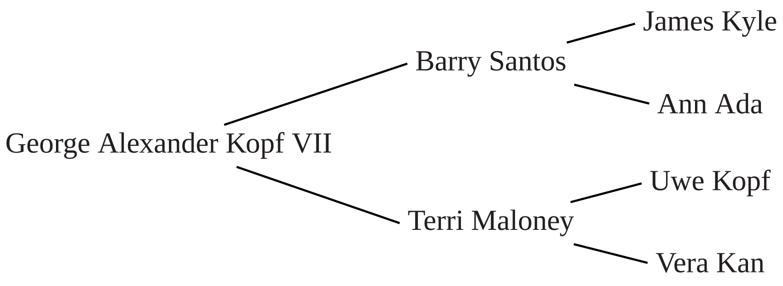
levelsep=*dim **Default: 2cm**

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 LaTeX, 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.

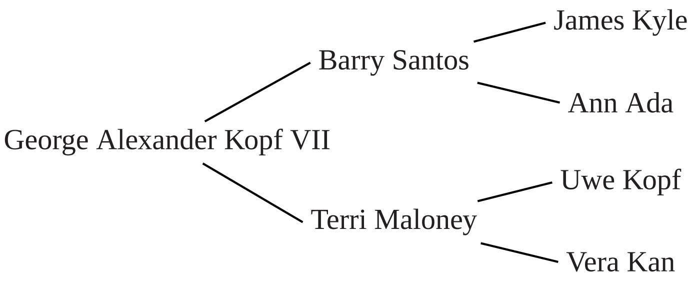
You are most likely to want to set **varlevelsep** to true in horizontal trees. Compare the following example:

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



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

```
\def\psedge#1#2{\ncdiagg[nodesep=3pt,angleA=180,armA=0]{#2}{#1}}
\pstree[treemode=R,levelsep=3cm]
  {\Tr{George Alexander Kopf VII}}{%
  \pstree[ref=c]{\Tr{Barry  Santos}}{\Tr{James  Kyle}  \Tr{Ann  Ada}}
  \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,

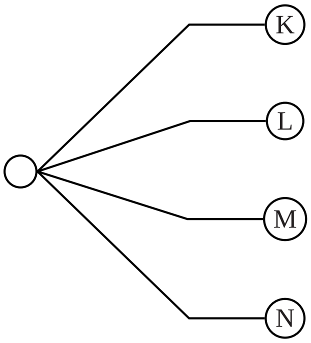
```
\ncline{\pspred}{\pssucc}
```

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

```
\psedge{\pspred}{\pssucc}
```

The default definition of \psedge is \ncline, but you can redefine it as you please with \def or LaTeX'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]



```
\def\psedge{\ncdiag[armA=0,angleB=180,armB=1]}
\pstree[treemode=R,levelsep=3.5,framesep=2pt]
   {\Tc{6pt}}{%
      \small  \Tcircle{K}  \Tcircle{L}  \Tcircle{M}  \Tcircle{N}}
```

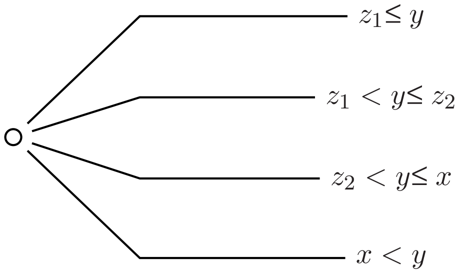
---

[28] LaTeX 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:

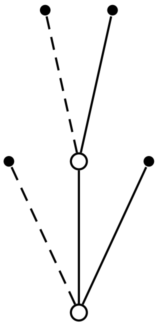
```
$
\def\psedge#1#2{%
   \ncdiagg[angleA=180,armA=-3,nodesep=4pt]{#2}{#1}}
% Or: \renewcommand{\psedge}[2]{ ... }
\pstree[treemode=R,levelsep=5]{\Tc{3pt}}{%
   \Tr{z_1\leq y}
   \Tr{z_1<y\leq z_2}
   \Tr{z_2<y\leq x}
   \Tr{x<y}}
$
```



Another way to define **\psedge** is with the

edge=*command* **Default: \ncline**

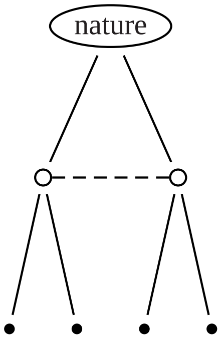
parameter. Be sure to enclose the value in braces {} if it contains commas or other parameter delimiters. This gets messy if your command is long, and you can't use arguments like in the preceding example, but for simple changes it is useful. For example, if I want to switch between a few node connections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



```
\def\dedge{\ncline[linestyle=dashed]}
\pstree[treemode=U,radius=2pt]{\Tc{3pt}}{%
   \TC*[edge=\dedge]
   \pstree{\Tc{3pt}}{\TC*[edge=\dedge] \TC*}
   \TC*}
```

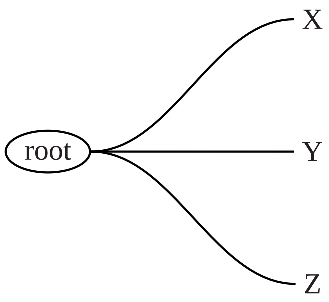
You can also set **edge=none** to suppress the node connection.

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:

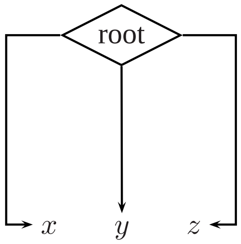


```
\pstree[nodesep=3pt,radius=2pt]{\Toval{nature}}{%
   \pstree{\Tc[name=top]{3pt}}{\TC*  \TC*}
   \pstree{\Tc[name=bot]{3pt}}{\TC*  \TC*}}
 \ncline[linestyle=dashed]{top}{bot}
```

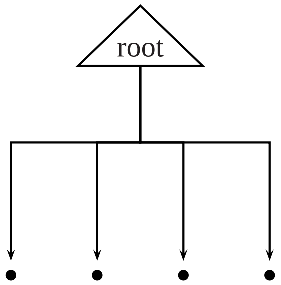
We conclude with the more examples.



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



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

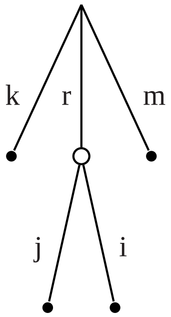


```
\psset{arrows=<-,armB=1,levelsep=3,treesep=1,
   angleB=-90,angleA=90,nodesepA=3pt}
\def\psedge#1#2{\ncangle{#2}{#1}}
\pstree[radius=2pt]{\Ttri{root}}{\TC*  \TC*  \TC*  \TC*}
```

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.



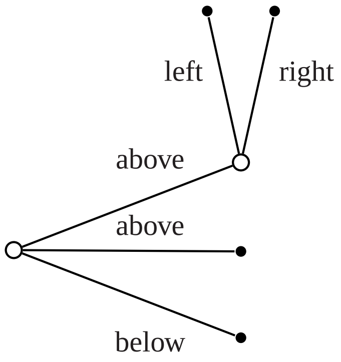
```
\pstree[radius=2pt]{\Tp}{%
   \psset{tpos=0.6}
   \TC*  \tlput{k}
   \pstree{\Tc{3pt}  \tlput[labelsep=3pt]{r}}{%
      \TC*  \tlput{j}
      \TC*  \trput{i}}
   \TC*  \trput{m}}
```

Within trees, the **tpos** parameter measures this distance from the predecessor to the successor, whatever the orientation of the true. (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 true:

	Short for:	
Char.	*Vert.*	*Horiz.*
^	**\tlput**	**\taput**
_	**\trput**	**\tbput**

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

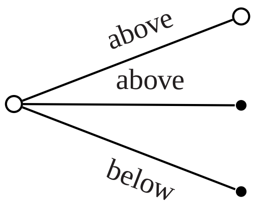


```
\psset{tpos=0.6}
\pstree[treemode=R,thistreesep=1,thislevelsep=3,
   radius=2pt]{\Tc{3pt}}{%
   \pstree[treemode=U,xbbr=20pt]{\Tc{3pt}^{above}}{%
      \TC*^{left}
      \TC*_{right}}
   \TC*^{above}
   \TC*_{below}}
```

You can change the character abbreviations with

\MakeShortTab{*char1*}{*char2*}

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



```
\psset{npos=0.6,nrot=:U}
\pstree[treemode=R,thistreesep=1,thislevelsep=3cm]{\Tc{3pt}}{%
   \Tc{3pt}\naput{above}
   \Tc*{2pt}\naput{above}
   \Tc*{2pt}\nbput{below}}
```

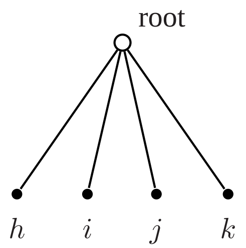
You can put labels on the nodes using **\nput**. However, **\pstree** won't take these labels into account when calculating the bounding boxes.

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:



```
\pstree[radius=2pt]{\Tc{3pt}\nput{45}{\pssucc}{root}}{%
   \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.,

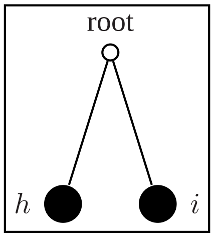
```
\MakeShortTnput{\tnput}
```

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 be setting several parameters.

To position the label on any side of the node (left, right, above or below), set:

tnpos=*l/r/a/b* **Default:**



```
\psframebox{%
  \pstree{\Tc{3pt}~[tnpos=a,tndepth=0,radius=4pt]{root}}{%
    \TC*~[tnpos=l]{$h$}
    \TC*~[tnpos=r]{$i$}}}
```

When you leave the argument empty, which is the default, PSTricks chooses the label position is 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 TEX 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.

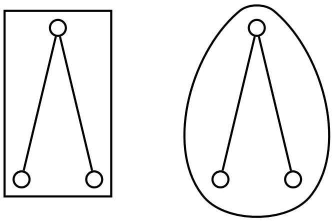
When labels are positioned on the left or right, the right or left edge of the label is positioned distance **tnsep** from the node. The vertical point that is aligned with the center of the node is set by

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.

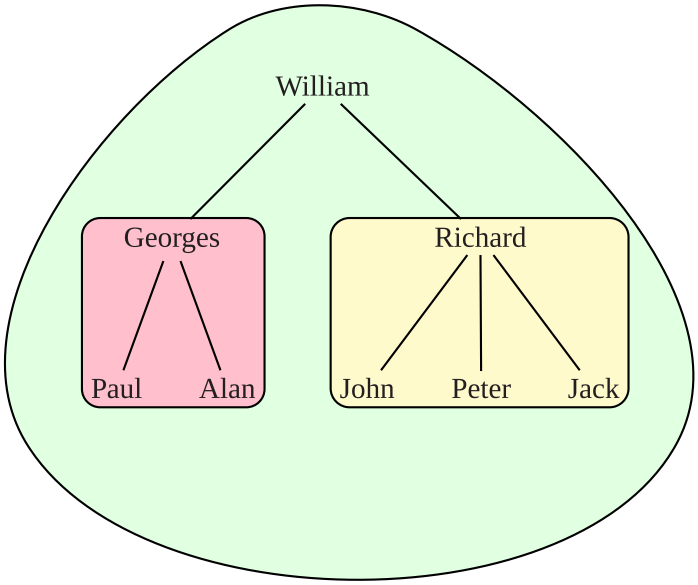
44 Framing

The **\pstreeframe** and **\pstreecurve** macros allow to frame trees or subtrees:



```
\pstreeframe{\pstree{\Tc{3pt}}{\Tc{3pt}\Tc{3pt}}}
\hspace{1cm}
\pstreecurve{\pstree{\Tc{3pt}}{\Tc{3pt}\Tc{3pt}}}
```

```
\pstreecurve[fillstyle=solid,fillcolor=PaleGreen,framesep=0.6]{%
   \pstree[nodesep=3pt,framearc=0.2]{\TR{William}}{%
      \pstreeframe[fillstyle=solid,fillcolor=Pink]{%
         \pstree{\TR{Georges}}{\TR{Paul}\TR{Alan}}}
      \pstreeframe[fillstyle=solid,fillcolor=LemonChiffon]{%
         \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:

bbl=*dim* **Default:**
bbr=*dim* **Default:**
bbh=*dim* **Default:**
bbd=*dim* **Default:**
xbbl=*dim* **Default: 0**
xbbr=*dim* **Default: 0**
xbbh=*dim* **Default:**
xbbd=*dim* **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, **l**eft, **r**ight, **h**eight, and **d**epth.

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).

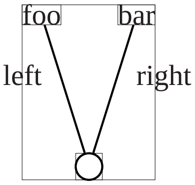
Save any fiddling with the bounding box until you are otherwise finished with the tree.

You can see the bounding boxes by setting the

showbbox=*true/false* **Default: false**

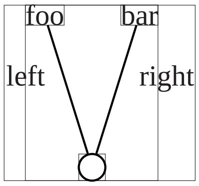
parameter to true. To see the bounding boxes of all the nodes in a tree, you have to set this parameter before the tree.

In the following example, the labels stick out of the bounding box:



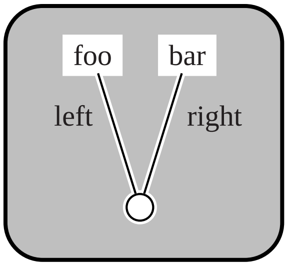
```
\psset{tpos=0.6,showbbox=true}
\pstree[treemode=U]{\Tc{5pt}}{%
   \TR{foo}^{left}
   \TR{bar}_{right}}
```

Here is how we fix it:



```
\psset{tpos=0.6,showbbox=true}
\pstree[treemode=U,xbbl=8pt,xbbr=14pt]{\Tc{5pt}}{%
   \TR{foo}^{left}
   \TR{bar}_{right}}
```

Now we can frame the tree:



```
\psframebox[fillstyle=solid,fillcolor=lightgray,framesep=14pt,
   linearc=14pt,cornersize=absolute,linewidth=1.5pt]{%
   \psset{tpos=0.6,border=1pt,nodesepB=3pt}
   \pstree[treemode=U,xbbl=8pt,xbbr=14pt]{%
      \Tc[fillcolor=white,fillstyle=solid]{5pt}}{%
      \TR*{foo}^{left}
      \TR*{bar}_{right}}}
```

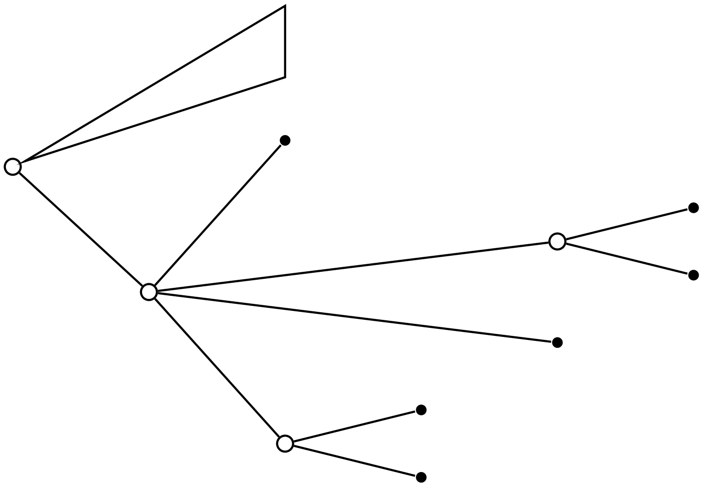
We would have gotten the same result by changing the bounding box of the two terminal nodes.

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.

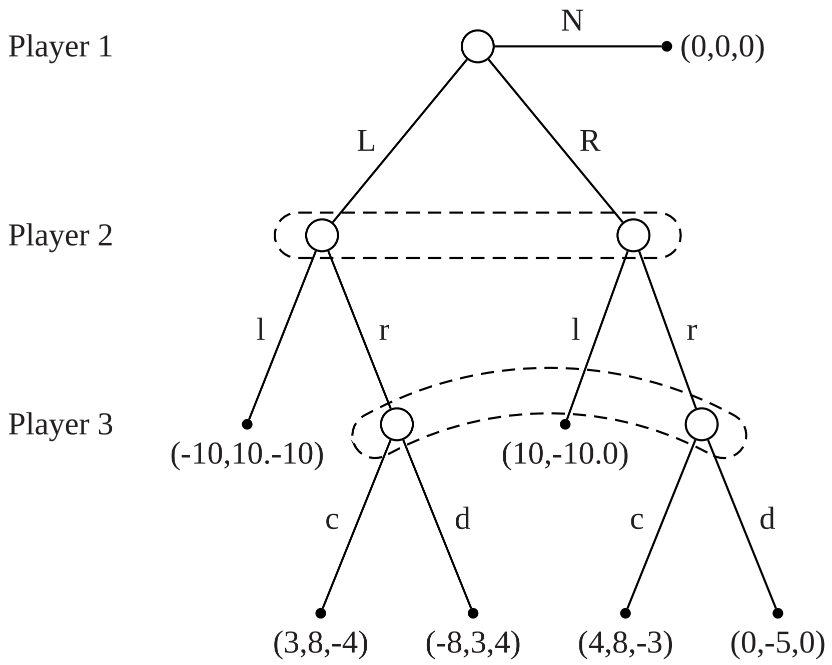
```
\pstree[treemode=R,levelsep=1.8,radius=2pt]{\Tc{3pt}}{%
   \skiplevel{\Tfan}
   \pstree{\Tc{3pt}}{%
      \TC*
      \skiplevels{2}
         \pstree{\Tc{3pt}}{\TC*  \TC*}
         \TC*
      \endskiplevels
      \pstree{\Tc{3pt}}{\TC*  \TC*}}}
```



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**.

```
\large
\psset{radius=6pt,dotsize=4pt}
```

```
\pstree[thislevelsep=0,edge=none,levelsep=2.5cm]{\Tn}{%
  \pstree{\TR{Player 1}}{\pstree{\TR{Player 2}}{\TR{Player 3}}}
  \psset{edge=\ncline}
  \pstree
    {\pstree[treemode=R]{\TC}{\Tdot ~{(0,0,0)} ^{N}}}{%
       \pstree{\TC[name=A] ^{L}}{%
         \Tdot ~{(-10,10.-10)} ^{l}
         \pstree{\TC[name=C] _{r}}{%
           \Tdot ~{(3,8,-4)} ^{c}
           \Tdot ~{(-8,3,4)} _{d}}}
       \pstree{\TC[name=B] _{R}}{%
         \Tdot ~{(10,-10.0)} ^{l}
         \pstree{\TC[name=D]_{r}}{%
           \Tdot ~{(4,8,-3)} ^{c}
           \Tdot ~{(0,-5,0)} _{d}}}}}
\ncbox[linearc=0.3,boxsize=0.3,linestyle=dashed,nodesep=0.4]{A}{B}
\ncarcbox[linearc=0.3,boxsize=0.3,linestyle=dashed,arcangle=25,
  nodesep=0.4]{D}{C}
```



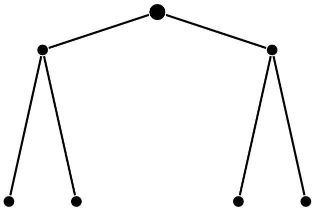
46 The scope of parameter changes

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* **Default:**
thistreenodesize=*dim* **Default:**
thistreefit=*tight/loose* **Default:**
thislevelsep=*dim* **Default:**

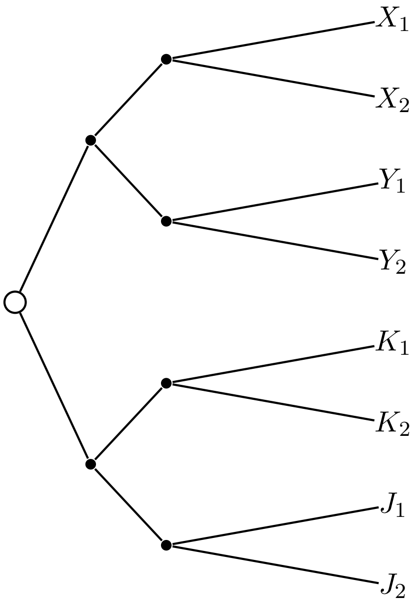
For example:



```
\pstree[thislevelsep=0.5,thistreesep=2,radius=2pt]{\Tc*{3pt}}{%
      \pstree{\TC*}{\TC*  \TC*}
      \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:

```
\[
   \def\pstreehookiii{\psset{thislevelsep=3cm}}
   \pstree[treemode=R,levelsep=1cm,radius=2pt]{\Tc{4pt}}{%
      \pstree{\TC*}{%
         \pstree{\TC*}{\Tr{X_1}  \Tr{X_2}}
         \pstree{\TC*}{\Tr{Y_1}  \Tr{Y_2}}}
      \pstree{\TC*}{%
         \pstree{\TC*}{\Tr{K_1}  \Tr{K_2}}
         \pstree{\TC*}{\Tr{J_1}  \Tr{J_2}}}}
\]
```



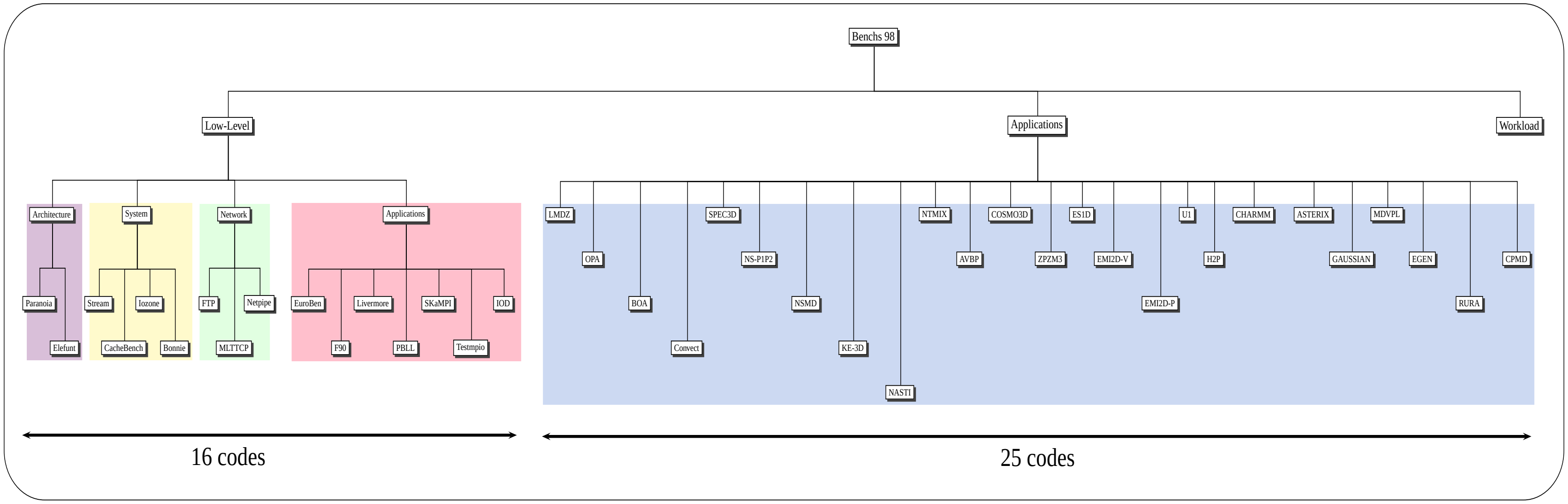
47 Complex examples

As trees are a very often representation structure used in many areas, we give here three more complex examples.

First a general organization tree:

```
% Connection style
\def\psedge#1#2{\ncangle[angleA=90,angleB=-90]{#2}{#1}}

\def\DirBa#1{\Tr{\psshadowbox{\footnotesize #1}}}
\def\DirBx#1#2{%
  \pstree[thislevelsep=#2,treenodesize=0.6]{\Tn}{\DirBa{#1}}}
\def\DirBb#1{\DirBx{#1}{1.5}}
\def\DirBc#1{\DirBx{#1}{3}}
\def\DirBd#1{\DirBx{#1}{4.5}}
\def\DirBe#1{\DirBx{#1}{6}}

\psset{levelsep=3,armB=1.5}%

\rotateleft{%
  \scaleboxto(22,7){%
   \psframebox[framesep=0.8,framearc=0.2]{%
     \pstree[xbbd=2.5]{\DirBa{\large Benchs 98}}
           {\pstree[treenodesize=0.3]
                 {\DirBa{\large Low-Level}%
                    \nput[labelsep=10.5]{-90}{\pssucc}{%
                      \psline[linewidth=0.1]{<->}(-7,1)(13.5,1)
                      \Huge 16 codes}}
                 {\pstreeframe*[linecolor=Thistle]{%
```

```
        \pstree{\DirBa{Architecture}}
              {\DirBa{Paranoia}
               \DirBb{Elefunt}}}
      \pstreeframe*[linecolor=LemonChiffon]{%
        \pstree{\DirBa{System}}
              {\DirBa{Stream}
               \DirBb{CacheBench}
               \DirBa{Iozone}
               \DirBb{Bonnie}}}
      \pstreeframe*[linecolor=PaleGreen]{%
        \pstree{\DirBa{Network}}
              {\DirBa{FTP}
               \DirBb{MLTTCP}
               \DirBa{Netpipe}}}
      \pstreeframe*[linecolor=Pink]{%
        \pstree[treenodesize=0.6]{\DirBa{Applications}}
              {\DirBa{EuroBen}
               \DirBb{F90}
               \DirBa{Livermore}
               \DirBb{PBLL}
               \DirBa{SKaMPI}
               \DirBb{Testmpio}
               \DirBa{IOD}}}}
\pstree{\DirBa{\large Applications}%
          \nput[labelsep=10.5]{-90}{\pssucc}{%
            \psline[linewidth=0.1]{<->}(-19,1)(22,1)
            \Huge 25 codes}}
       {\pstreeframe*[linecolor=LightBlue]{%
          \pstree[thislevelsep=0]{\Tn}
                {\DirBa{LMDZ}
                 \DirBb{OPA}
                 \DirBc{BOA}
                 \DirBd{Convect}
                 \DirBa{SPEC3D}
                 \DirBb{NS-P1P2}
                 % ...
                 \DirBb{CPMD}}}}
\tspace{17}
\DirBa{\large Workload}}}}}
```

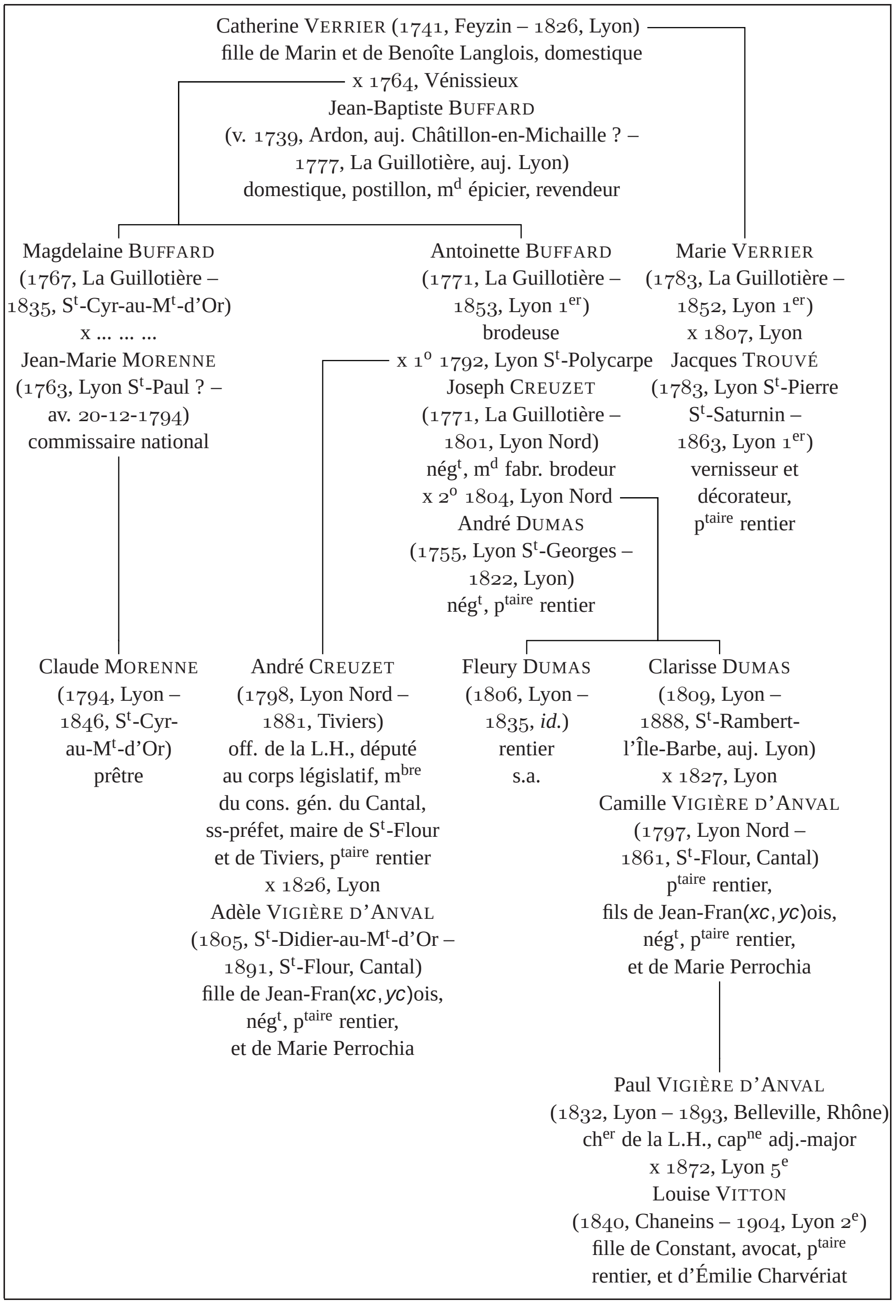

Benchs 98
Low-Level
Applications
Workload
Architecture
Paranoia
Elefunt
System
Stream
CacheBench
Iozone
Bonnie
Network
FTP
MLTTCP
Netpipe
Applications
EuroBen
F90
Livermore
PBLL
SKaMPI
Testmpio
IOD
LMDZ
OPA
BOA
Convect
SPEC3D
NS-P1P2
NSMD
KE-3D
NASTI
NTMIX
AVBP
COSMO3D
ZPZM3
ES1D
EMI2D-V
EMI2D-P
U1
H2P
CHARMM
ASTERIX
GAUSSIAN
MDVPL
EGEN
RURA
CPMD
16 codes
25 codes

And, second, an example of a complex genealogical tree, coming from Françoise Coustillas and Louis Rigot.

```
% Defaut edge
\renewcommand{\psedge}{\ncangle[angleA=-90]}

% Macro to define one person
\newcommand{\Person}[3][]{%
\TR[#1]{%
\setlength{\tabcolsep}{0mm}%
\begin{tabular}[t]{#2}#3\end{tabular}}}

% Definition of people
\newcommand{\CatherineVerrier}{%
\Person{c}{%
  \rnode{NodeTempA}{%
    Catherine V\textsc{errier} (\oldstylenums{1741}, Feyzin –
    \oldstylenums{1826}, Lyon)\hspace{1mm}}\\
  fille de Marin et de Beno^^eete Langlois, domestique\\
  \rnode{NodeTempB}{%
    \hspace{1mm}x \oldstylenums{1764}, V^^e9nissieux}\\
  Jean-Baptiste B\textsc{uffard}\\
  (v. \oldstylenums{1739}, Ardon,
   auj. Ch^^e2tillon-en-Michaille ? –\\
   \oldstylenums{1777}, La Guilloti^^e8re, auj. Lyon)\\
  domestique, postillon, m\textsuperscript{d} ^^e9picier, revendeur}}

\newcommand{\MarieVerrier}{%
\Person[edge={\ncangle[angleA=0]}]{c}{%
  Marie V\textsc{errier}\\
   % ...

\newcommand{\MadeleineBuffard}{%
\Person[edge={\ncangles[angleA=180,armA=2.2cm]}]{c}{%
  Magdelaine B\textsc{uffard}\\
% ...
\scriptsize

{\Large Parent^^e9 avec Paul Vigi^^e8re d’Anval}
{\Large des parties signataires de son acte de bapt^^eame}

\fbox{\makebox[\textwidth]{%
  \pstree[levelsep=*1.5\baselineskip,treesep=-0.1,
          armB=0.6\baselineskip,angleB=90,linewidth=0.015]
         {\CatherineVerrier
          \gdef\pssucc{NodeTempB}}
         {\pstree{\MadeleineBuffard}
                 {\ClaudeMorenne}
          \pstree[thistreesep=1mm]
                 {\AntoinetteBuffard
                  \gdef\pssucc{NodeTempC}}
                 {\AndreCreuzet
```

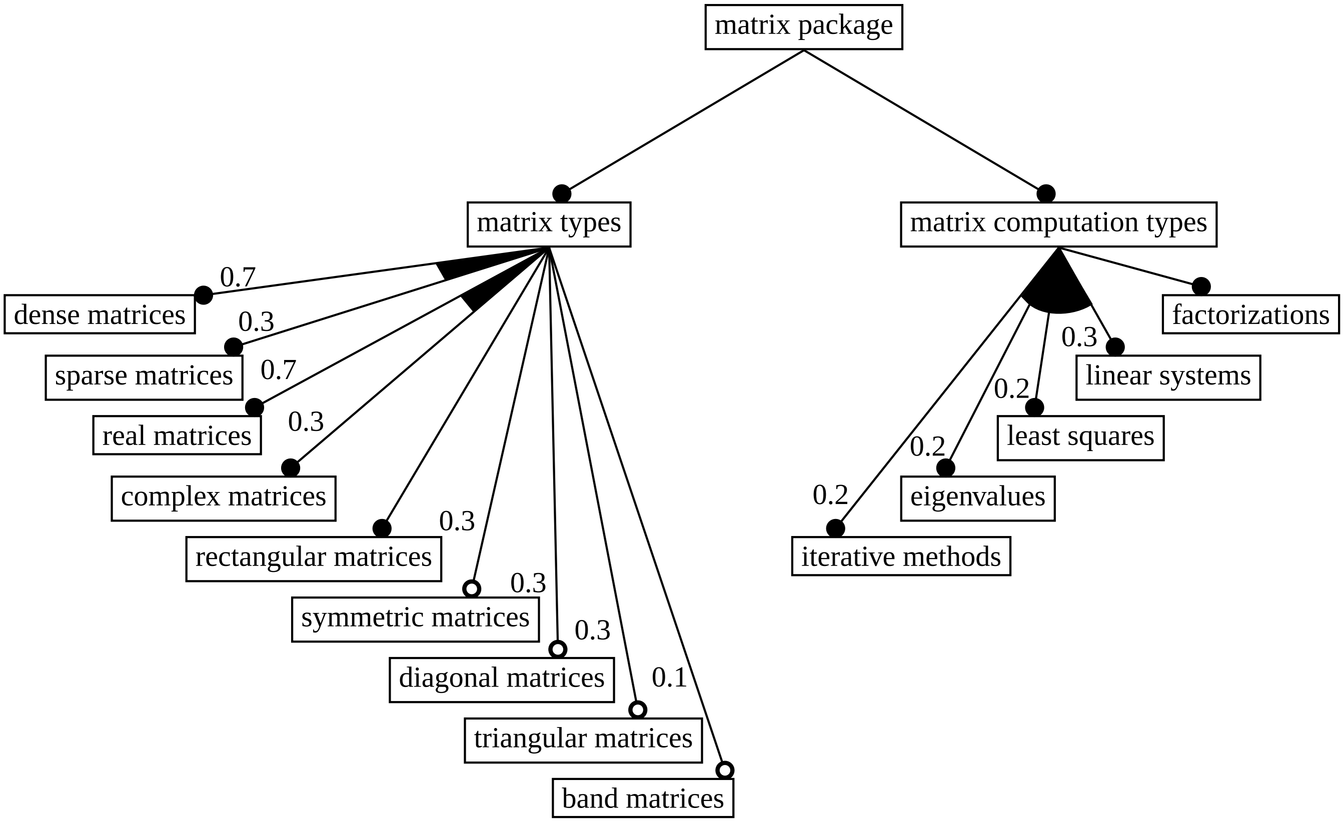
```
                    \gdef\pspred{NodeTempD}%
                    \FleuryDumas
                    \pstree[levelsep=*4\baselineskip]
                           {\ClarisseDumas}
                           {\PaulVigiere}}
           \gdef\pspred{NodeTempA}%
           \MarieVerrier}}}
```

Parenté avec Paul Vigière d’Anval
des parties signataires de son acte de baptême


Catherine VERRIER (1741, Feyzin – 1826, Lyon)
fille de Marin et de Benoîte Langlois, domestique
x 1764, Vénissieux
Jean-Baptiste BUFFARD
(v. 1739, Ardon, auj. Châtillon-en-Michaille ? – 1777, La Guillotière, auj. Lyon)
domestique, postillon, m^{d} épicier, revendeur
Magdelaine BUFFARD
(1767, La Guillotière – 1835, S^{t}-Cyr-au-M^{t}-d’Or)
x … … …
Jean-Marie MORENNE
(1763, Lyon S^{t}-Paul ? – av. 20-12-1794)
commissaire national
Antoinette BUFFARD
(1771, La Guillotière – 1853, Lyon 1er)
brodeuse
x 1^{o} 1792, Lyon S^{t}-Polycarpe
Joseph CREUZET
(1771, La Guillotière – 1801, Lyon Nord)
nég^{t}, m^{d} fabr. brodeur
x 2^{o} 1804, Lyon Nord
André DUMAS
(1755, Lyon S^{t}-Georges – 1822, Lyon)
nég^{t}, p^{taire} rentier
Marie VERRIER
(1783, La Guillotière – 1852, Lyon 1er)
x 1807, Lyon
Jacques TROUVÉ
(1783, Lyon S^{t}-Pierre S^{t}-Saturnin – 1863, Lyon 1er)
vernisseur et décorateur,
p^{taire} rentier
Claude MORENNE
(1794, Lyon – 1846, S^{t}-Cyr-au-M^{t}-d’Or)
prêtre
André CREUZET
(1798, Lyon Nord – 1881, Tiviers)
off. de la L.H., député au corps législatif, m^{bre} du cons. gén. du Cantal, ss-préfet, maire de S^{t}-Flour et de Tiviers, p^{taire} rentier
x 1826, Lyon
Adèle VIGIÈRE D’ANVAL
(1805, S^{t}-Didier-au-M^{t}-d’Or – 1891, S^{t}-Flour, Cantal)
fille de Jean-Fran(xc,yc)ois, nég^{t}, p^{taire} rentier, et de Marie Perrochia
Fleury DUMAS
(1806, Lyon – 1835, *id.*)
rentier
s.a.
Clarisse DUMAS
(1809, Lyon – 1888, S^{t}-Rambert-l’Île-Barbe, auj. Lyon)
x 1827, Lyon
Camille VIGIÈRE D’ANVAL
(1797, Lyon Nord – 1861, S^{t}-Flour, Cantal)
p^{taire} rentier,
fils de Jean-Fran(xc,yc)ois, nég^{t}, p^{taire} rentier, et de Marie Perrochia
Paul VIGIÈRE D’ANVAL
(1832, Lyon – 1893, Belleville, Rhône)
ch^{er} de la L.H., cap^{ne} adj.-major
x 1872, Lyon 5^{e}
Louise VITTON
(1840, Chaneins – 1904, Lyon 2^{e})
fille de Constant, avocat, p^{taire} rentier, et d’Émilie Charvériat

```
\def\MyNodeA{\@ifnextchar[{\MyNodeA@i}{\MyNodeA@i[]}}
\def\MyNodeA@i[#1]#2{\Tr[#1]{\psframebox{#2}}}
\def\MyNodeB{\@ifnextchar[{\MyNodeB@i}{\MyNodeB@i[]}}
\def\MyNodeB@i[#1]#2{\TR[#1]{\psframebox{#2}}}
\def\MyNodeC{\@ifnextchar[{\MyNodeC@i}{\MyNodeC@i[]}}
\def\MyNodeC@i[#1]#2#3{\Tr[#1]{\rnode[b]{#2}{\psframebox{#3}}}}

\SpecialCoor

\pstree[arrows={-*},arrowscale=2,nodesepB=0.1]
     {\MyNodeA[ref=b]{matrix  package}}
     {\pstree[treesep=-2,levelsep=1.5]
           {\MyNodeC[ref=t]{Types}{matrix  types}}
           {\psset{levelsep=0.8,labelsep=0.1}%
            \def\pspred{Types}%
            \MyNodeB[href=-0.4]{%
                \Rnode[href=-0.4]{Dense}{dense  matrices}}
              \tlput[tpos=0.44]{0.7}
            \skiplevels{1}
              \MyNodeB{\Rnode{Sparse}{sparse  matrices}}
                \tlput[tpos=0.58]{0.3}
            \endskiplevels
            \pspolygon*([nodesep=3.3]{\pspred}Dense)(\pspred)
                     ([nodesep=2.8]{\pspred}Sparse)
            \skiplevels{2}
              \MyNodeB[href=0.3]{%
                \Rnode[href=0.3]{Real}{real  matrices}}
                  \tlput[tpos=0.65]{0.7}
            \endskiplevels
            \skiplevels{3}
              \MyNodeB[href=0.3]{%
                  \Rnode[href=0.3]{Complex}{complex  matrices}}
                \tlput[tpos=0.7]{0.3}
            \endskiplevels
            \pspolygon*([nodesep=2.75]{\pspred}Real)(\pspred)
                     ([nodesep=2.3]{\pspred}Complex)
            \skiplevels{4}
              \MyNodeB[href=0.4]{rectangular  matrices}
            \endskiplevels
            \psset{arrows={-o},tpos=0.78}%
            \skiplevels{5}
              \MyNodeB[href=0.4]{symmetric  matrices}
                \tlput[tpos=0.74]{0.3}
            \endskiplevels
            \skiplevels{6}
              \MyNodeB[href=0.5]{diagonal  matrices}
                \tlput{0.3}
            \endskiplevels
            \skiplevels{7}
              \MyNodeB[href=0.5]{triangular  matrices}
                \tlput{0.3}
            \endskiplevels
```

```
        \skiplevels{8}
          \MyNodeB[href=1]{band  matrices}
            \tlput[labelsep=0]{0.1}
        \endskiplevels
        \tspace{6}\Tn}
\pstree[treesep=-3.5,levelsep=1.5]
       {\MyNodeC[ref=t]{Computation}{%
          matrix  computation  types}}
       {\def\pspred{Computation}%
        \MyNodeB[href=0.6]{factorizations}
        \psset{levelsep=0.8,labelsep=0.05,tpos=0.8}%
        \skiplevels{1}
          \MyNodeB[href=-0.4]{%
              \Rnode[href=-0.4]{Linear}{linear  systems}}
            \tlput[tpos=0.7]{0.3}
        \endskiplevels
        \skiplevels{2}
          \MyNodeB[href=-0.6]{least  squares}
            \tlput[tpos=0.75]{0.2}
        \endskiplevels
        \skiplevels{3}
          \MyNodeB[href=-0.6]{eigenvalues}
            \tlput{0.2}
        \endskiplevels
        \skiplevels{4}
          \MyNodeB[href=-0.8]{%
              \Rnode[href=-0.8]{Iterative}{iterative  methods}}
            \tlput{0.2}
        \endskiplevels
        \pnode([nodesep=0.8]{\pspred}Linear){LinearB}
        \pnode([nodesep=3.3]{\pspred}Iterative){IterativeB}
        \pscustom*[arrows=-]{%
          \psline(IterativeB)(\pspred)(LinearB)
          \ncarc[arcangle=40]{LinearB}{IterativeB}}}}
```


matrix package
matrix types
matrix computation types
0.7
dense matrices
0.3
sparse matrices
0.7
real matrices
0.3
complex matrices
rectangular matrices
0.3
symmetric matrices
0.3
diagonal matrices
0.3
triangular matrices
0.1
band matrices
factorizations
0.3
linear systems
0.2
least squares
0.2
eigenvalues
0.2
iterative methods

IX

Filling and Tiling[29]

48 Overview

pst-fill

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:

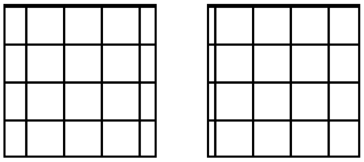
```
\def\Square{%
\pspicture(0.5,0.5)\psframe[dimen=middle](0.5,0.5)\endpspicture}
```

---

[29] This chapter is an adaptation of the documentation for this package written in 1997 by Denis Girou.

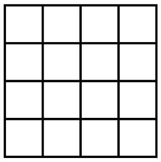
[30] For an extensive presentation of tilings, in their history and usage in many fields, see the reference book by Branko Grünbaum and Geoffrey Shephard, *Tilings and Patterns* [28].

the difference between the two modes is shown here; *filling*:



```
\psboxfill{\Square}

\begin{pspicture}(2,2)
   \psframe[fillstyle=boxfill](2,2)
\end{pspicture}
\hspace{5mm}
\begin{pspicture}(2,2)
   \psframe[fillstyle=boxfill](2,2)
\end{pspicture}
```

where, as you can see, the initial position is arbitrary and depends on the current point, and *tiling*:



```
\psboxfill{\Square}
\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, LaTeX users must load the package with the *tiling* parameter and plain TeX and ConTeXt 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:

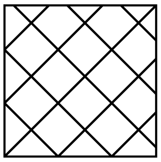
```
\def\Tiling#1#2{%
\edef\Temp{#1}%
\pspicture#2
    \ifx\Temp\empty
        \psframe[fillstyle=boxfill]#2
    \else
        \psframe[fillstyle=boxfill,#1]#2
    \fi
\endpspicture}
```

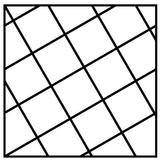
There are several parameters available to change the way the filling/tiling is defined, and one debugging option.

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.





```
\psboxfill{\Square}

\Tiling{fillangle=45}{(2,2)}
\hspace{5mm}
\Tiling{fillangle=-60}{(2,2)}
```

fillsepx=*dim* **Default: 0pt**

The value of the horizontal separation between consecutive patterns.[31]

fillsepy=*dim* **Default: 0pt**

The value of the vertical separation between consecutive patterns.[31]

fillsep=*dim* **Default: 0pt**

The value of the horizontal and vertical separations between consecutive patterns.[32]

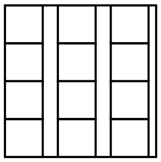
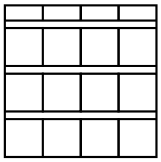
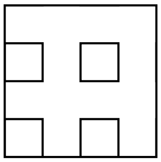
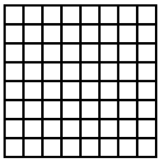
These values can be negative, which allow the tiles to overlap.

---

[31] This option is only available in the *tiling* mode.

[32] More precisely, the default value is 0pt in *tiling* mode and 2pt in *filling* mode.

```
\psboxfill{\Square}
\Tiling{fillsepx=2mm}{(2,2)}\hfill
\Tiling{fillsepy=1mm}{(2,2)}\hfill
\Tiling{fillsep=0.5}{(2,2)}\hfill
\Tiling{fillsep=-0.25}{(2,2)}
```

fillcyclex=*num* **Default: 0**

The shift coefficient applied to each row.[31]

fillcycley=*num* **Default: 0**

The shift coefficient applied to each collumn.[31]

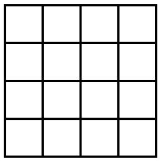
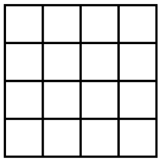
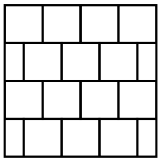
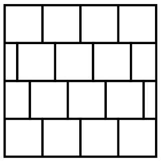
fillcycle=*num* **Default: 0**

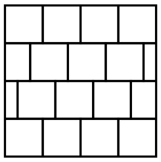
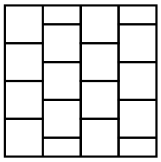
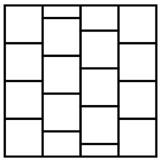
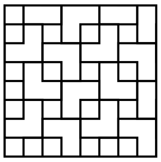
The shift coefficient applied both to each row and each collumn.

For instance, if **fillcyclex** 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 **fillcyclex** is 3, etc.

These values must be integers and can be negative.

```
\psboxfill{\Square}

\Tiling{}{(2,2)}\hfill
\Tiling{fillcyclex=1}{(2,2)}\hfill
\Tiling{fillcyclex=2}{(2,2)}\hfill
\Tiling{fillcyclex=3}{(2,2)}

\vspace{3mm}
\Tiling{fillcyclex=-3}{(2,2)}\hfill
\Tiling{fillcycley=2}{(2,2)}\hfill
\Tiling{fillcycley=-3}{(2,2)}\hfill
\Tiling{fillcycle=2}{(2,2)}\hfill
```

fillmovex=*dim* **Default: 0pt**

The value of the horizontal move between consecutive patterns.[31]

fillmovey=*dim* **Default: 0pt**

The value of the vertical move between consecutive patterns.[31]

fillmove=*dim* **Default: 0pt**

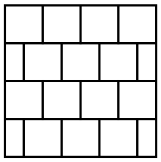
The value of the horizontal and vertical move between consecutive patterns.[31]

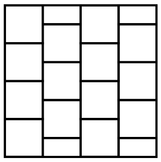
These parameters allow the patterns to overlap and to draw some special kinds of tilings.

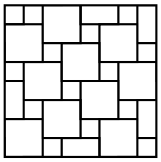
Their values can be negative.

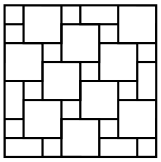
In some cases, the effect of these parameters will be the same as that with the fillcycle? ones, but this is not true for all values.

```
\psboxfill{\Square}
\Tiling{fillmovex=0.25}{(2,2)}\hfill
\Tiling{fillmovey=0.25}{(2,2)}\hfill
\Tiling{fillmove=0.25}{(2,2)}\hfill
\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).

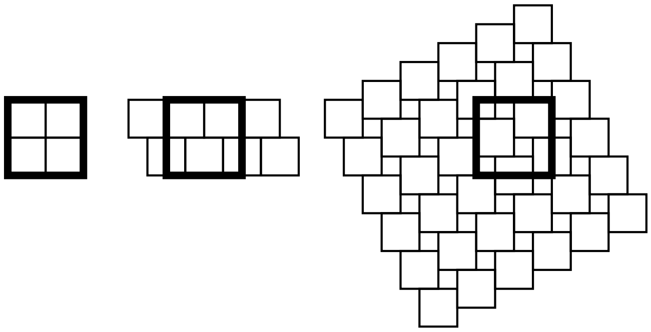
PstDebug=*num* **Default: 0**

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.

```
\psboxfill{\Square}
\psset{unit=0.5,dimen=middle,linewidth=1mm,PstDebug=1}
\Tiling{}{(2,2)}\hspace{1cm}
\Tiling{fillcyclex=2}{(2,2)}\hspace{3cm}
\Tiling{fillmove=0.5}{(2,2)}
```



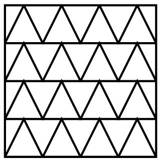
50 Examples

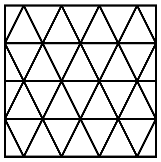
The single **\psboxfill** macro has many variations and different uses. We will try here to demonstrate many of them:

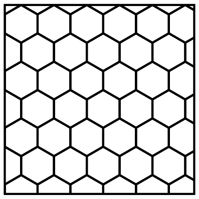
50.1 Kind of tiles

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 Archimedian 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.

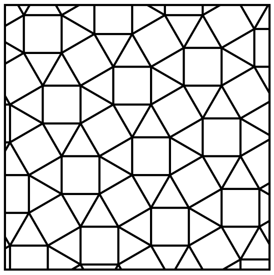
```
\def\Triangle{%
\pspicture(1,1)\pstriangle[dimen=middle](0.5,0)(1,1)\endpspicture}
\def\Hexagon{%
\pspicture(0.866,0.75)%  sin(60)=0.866
   \SpecialCoor
   \pspolygon[dimen=middle](0.5;30)(0.5;90)(0.5;150)
                                  (0.5;210)(0.5;270)(0.5;330)%Hexagon
\endpspicture}

\psset{unit=0.5}
\psboxfill{\Triangle}
\Tiling{}{(4,4)}\hfill
% The two other regular tilings
\Tiling{fillcyclex=2}{(4,4)}\hfill
\psboxfill{\Hexagon}
\Tiling{fillcyclex=2,fillloopaddy=1}{(5,5)}
```

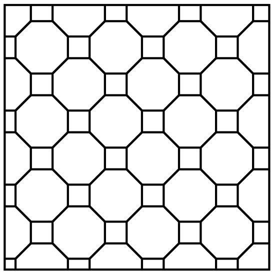






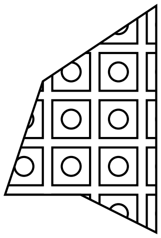
```
\def\ArchimedianA{% Archimedian tiling 3^2.4.3.4
\psset{dimen=middle}%
\pspicture(1.866,1.866)% sin(60)=0.866
   \psframe(1,1)
   \psline(1,0)(1.866,0.5)(1,1)
          (0.5,1.866)(0,1)(-0.866,0.5)
   \psline(0,0)(0.5,-0.866)
\endpspicture}

\def\ArchimedianB{% Archimedian tiling 4.8^2
\psset{dimen=middle,unit=1.5}%
\pspicture(1.3066,0.6533)% cos(22.5) + sin(22.5) = 1.3066
                         % cos(22.5) - sin(22.5) = 0.6533
   \SpecialCoor
   \pspolygon(0.5;22.5)(0.5;67.5)(0.5;112.5)(0.5;157.5)(0.5;202.5)
             (0.5;247.5)(0.5;292.5)(0.5;337.5)% Octogon
\endpspicture}

\psset{unit=0.5}
\psboxfill{\ArchimedianA}
\Tiling{fillmove=0.5}{(7,7)}\hfill
\psboxfill{\ArchimedianB}
\Tiling{fillcyclex=2,fillloopaddy=1}{(7,7)}
```

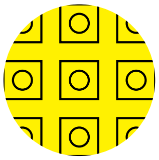




We can of course tile an arbitrarily defined area; with the **addfillstyle** parameter, we can easily mix the boxfill style with another one.

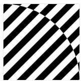
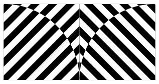
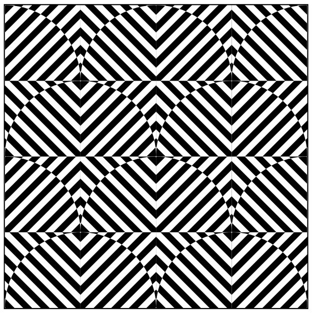
```
\psset{unit=0.5,dimen=middle}
\psboxfill{%
   \pspicture(1,1)
      \psframe(1,1)
      \pscircle(0.5,0.5){0.25}
   \endpspicture}

\begin{pspicture}(4,6)
   \pspolygon[fillstyle=boxfill,fillsep=0.25](0,1)(1,4)(4,6)(4,0)(2,1)
\end{pspicture}
\hspace{2cm}
\begin{pspicture}(4,4)
  \pscircle[linestyle=none,fillstyle=solid,fillcolor=yellow,fillsep=0.5,
     addfillstyle=boxfill](2,2){2}
\end{pspicture}
```





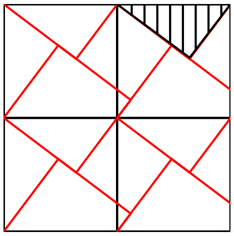
Various effects can be obtained; sometimes complicated ones are surprisingly easy, as in this example reproduced from one by Slavik Jablan in the field of *OpTiles*, inspired by *Op-art*:

```
\def\ProtoTile{%
\pspicture(1,1)
  \psset{linestyle=none,linewidth=0,hatchwidth=0.08333\psunit,
     hatchsep=0.08333\psunit}%  1/12=0.08333
  \psframe[fillstyle=solid,fillcolor=black,addfillstyle=hlines,
     hatchcolor=white](1,1)
  \pswedge[fillstyle=solid,fillcolor=white,addfillstyle=hlines]{1}{0}{90}
\endpspicture}

\def\BasicTile{%
\pspicture(2,1)
  \rput[lb](0,0){\ProtoTile}
  \rput[lb](1,0){\rotateleft{\ProtoTile}}
\endpspicture}

\ProtoTile\hfill\BasicTile\hfill
\psboxfill{\BasicTile}
\Tiling{fillcyclex=2}{(4,4)}
```

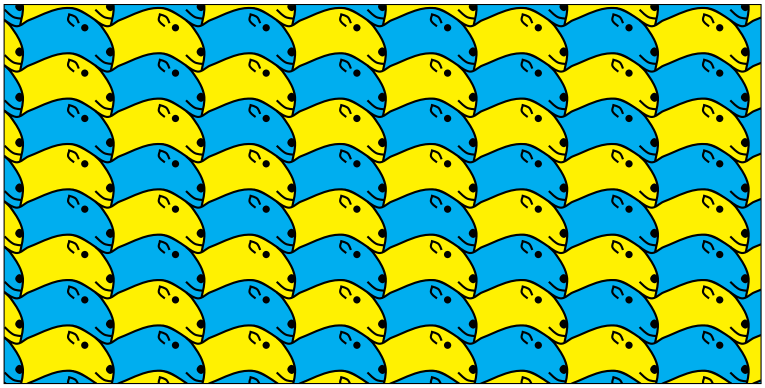
It is also possible to superimpose several different tilings. Here is the splendid visual proof of the Pythagore theorem done by the Arab mathematician Annairizi around the year 900, given by superposition of two tilings by squares of different sizes.

```
\psset{unit=1.5,dimen=middle}
\pspicture*(2,2)
  \psboxfill{\pspicture(1,1)
               \psframe(1,1)
              \endpspicture}
  \psframe[fillstyle=boxfill](2,2)
  \psboxfill{\pspicture(1,1)
               \rput{-37}{\psframe[linecolor=red](0.8,0.8)}
              \endpspicture}
  \psframe[fillstyle=boxfill](3,4)
  \pspolygon[fillstyle=hlines,hatchangle=90](1,2)(1.64,1.53)(2,2)
\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 crystalographer Jevgraf Fedorov at the end of the 19th century).

```
\def\SheepHead#1{%
\pspicture(3,1.5)
  \pscustom[liftpen=2,fillstyle=solid,fillcolor=#1]{%
    \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)
              (2.2,1.9)(3,1.5)(3,1.5)(3.2,1.3)(3.6,0.5)(3.4,-0.3)(3,0)
              (2.2,0.4)(0.5,-0.2)}
  \pscircle*(2.65,1.25){0.12\psunit}                 %  Eye
  \psccurve*(3.5,0.3)(3.35,0.45)(3.5,0.6)(3.6,0.4)%  Muzzle
  \pscurve(3,0.35)(3.3,0.1)(3.6,0.05)                %  Mouth
  \pscurve(2.3,1.3)(2.1,1.5)(2.15,1.7)
  \pscurve(2.1,1.7)(2.35,1.6)(2.45,1.4)              %  Ear
\endpspicture}

\psboxfill{\psset{unit=0.4}\SheepHead{yellow}\SheepHead{cyan}}
\Tiling{fillcyclex=2,fillloopadd=1}{(10,5)}
```

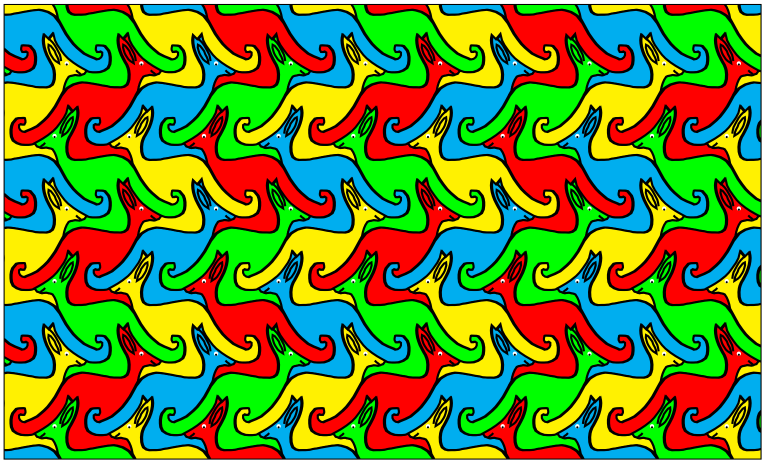


The next example shows a tiling of the *pg* category (the code for the kangaroo itself is too long to be shown here, but has no difficulties).[33]; the kangaroo is reproduced from an original picture by Raoul Raba and here is a translation into PSTricks from the one drawn by Emmanuel Chailloux and Guy Cousineau for their MLgraph system [6]

```
\psboxfill{%
  \psset{unit=0.4}%
  \Kangaroo{yellow}\Kangaroo{red}%
    \Kangaroo{cyan}\Kangaroo{green}%
  \scalebox{-1  1}{%
    \rput(1.235,4.8){%
      \Kangaroo{green}\Kangaroo{cyan}%
        \Kangaroo{red}\Kangaroo{yellow}}}}
\Tiling{fillloopadd=1}{(10,6)}
```

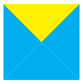
---

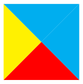
[33] You will find it in the source code of this documentation.

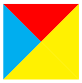


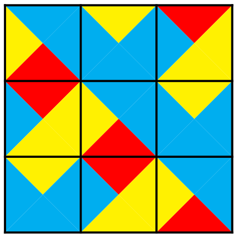
And now a Wang tiling [84], based on very simple tiles in the form of a square and composed of four colored triangles. Such tilings are simply built with a matching color constraint. Despite its simplicity, it is an important kind of tiling, as Wang and others used them to study the special class of *aperiodic* tilings, and also because it was shown that (surprisingly) this tiling is similar to a Turing machine.

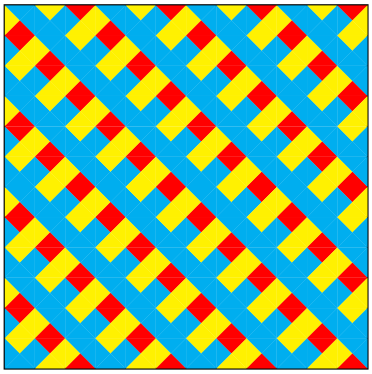
```
\def\WangTile#1#2#3#4{%
\pspicture(1,1)
  \pspolygon*[linecolor=#1](0,0)(0,1)(0.5,0.5)
  \pspolygon*[linecolor=#2](0,1)(1,1)(0.5,0.5)
  \pspolygon*[linecolor=#3](1,1)(1,0)(0.5,0.5)
  \pspolygon*[linecolor=#4](1,0)(0,0)(0.5,0.5)
\endpspicture}

\def\WangTileA{\WangTile{cyan}{yellow}{cyan}{cyan}}
\def\WangTileB{\WangTile{yellow}{cyan}{cyan}{red}}
\def\WangTileC{\WangTile{cyan}{red}{yellow}{yellow}}

\def\WangTiles#1{%
\pspicture(3,3)
  \psset{ref=lb}%
  \rput(0,2){\WangTileB}\rput(1,2){\WangTileA}\rput(2,2){\WangTileC}
  \rput(0,1){\WangTileC}\rput(1,1){\WangTileB}\rput(2,1){\WangTileA}
  \rput(0,0){\WangTileA}\rput(1,0){\WangTileC}\rput(2,0){\WangTileB}
  #1
\endpspicture}

\WangTileA\hfill\WangTileB\hfill\WangTileC\hfill
\WangTiles{\psgrid[subgriddiv=0,gridlabels=0](3,3)}

\vspace{2mm}
\psset{unit=0.4}
\psboxfill{\WangTiles{}}
\Tiling{}{(12,12)}
```







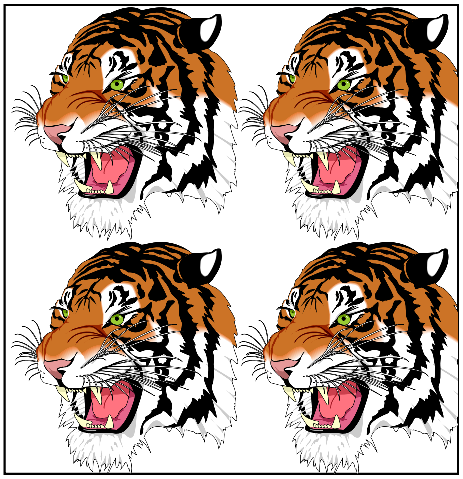




50.2 External graphic files

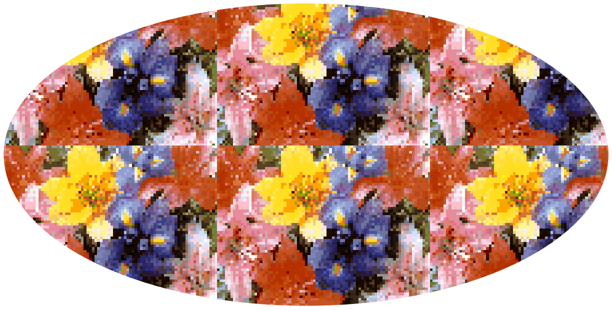
We can fill an arbitrary area with an external PostScript image. We have only, as usual, to worry about the *BoundingBox* definition if there is not one provided or if it is inaccurate, as in the case of the well known tiger picture (part of the Ghostscript distribution).

```
\psboxfill{%
  \raisebox{-1cm}{%
    \includegraphics[bb=17 176 562 740,width=3cm]{tiger}}}
\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…

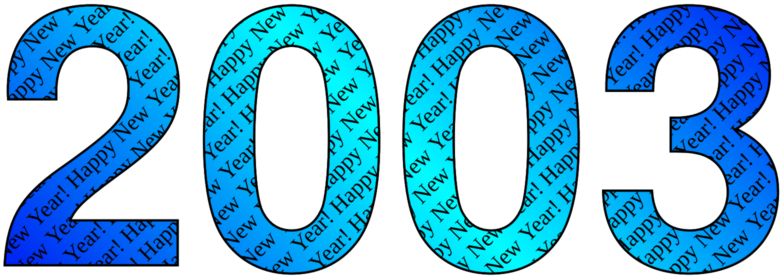
```
\psboxfill{\includegraphics{flowers}}
\pspicture(8,4)
  \psellipse[fillstyle=boxfill,fillsize={(8,4)}](4,2)(4,2)
\endpspicture
```



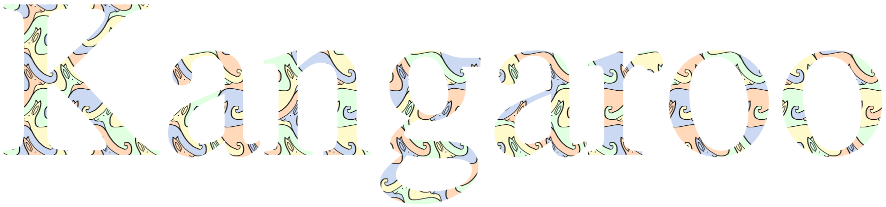
50.3 Tiling of characters

We can also use the **\psboxfill** macro to fill the interior of characters for special effects like the following:

```
\DeclareFixedFont{\Sf}{T1}{phv}{b}{n}{5cm}
\DeclareFixedFont{\Rm}{T1}{ptm}{m}{n}{3mm}
\psboxfill{\Rm Happy New Year!}
\pspicture*(10.8,3.8)
  \rput(5.4,0.1){%
    \pscharpath[fillstyle=gradient,gradangle=-45,gradmidpoint=0.5,
      addfillstyle=boxfill,fillangle=45,fillsep=0.7mm]
      {\rput[b](0,0){\Sf 2003}}}
\endpspicture
```



```
\DeclareFixedFont{\Rmm}{T1}{ptm}{m}{n}{3cm}
\psboxfill{%
 \psset{unit=0.2,linewidth=0.2pt}%
 \Kangaroo{PeachPuff}\Kangaroo{PaleGreen}%
 \Kangaroo{LightBlue}\Kangaroo{LemonChiffon}%
 \scalebox{-1 1}{%
   \rput(1.235,4.8){%
     \Kangaroo{LemonChiffon}\Kangaroo{LightBlue}%
     \Kangaroo{PaleGreen}\Kangaroo{PeachPuff}}}}
% A kangaroo of kangaroos...
\begin{pspicture}(11.6,2.7)
  \pscharpath[linestyle=none,fillloopadd=1,fillstyle=boxfill]
             {\rput[b](5.8,0){\Rmm Kangaroo}}
\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 it principle, it draws sophisticated curves with strange properties.

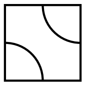
Unfortunately, the ‘pst-fill’ package does not work in a straightforward manner, because the **\psboxfill** macro stores the content of the tile in a TEX box, which is static. So the call of the random function is done only once, which explains why only one rotation of the tile is used for all the tiling. Only one of the two rotations can differ from one drawing to the next…

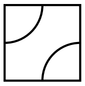
```
\def\ProtoTileLPT{{% LPT prototile
\psset{dimen=middle}%
\pspicture(1,1)
  \psframe(1,1)
  \psarc(0,0){0.5}{0}{90}
  \psarc(1,1){0.5}{-180}{-90}
\endpspicture}}
```

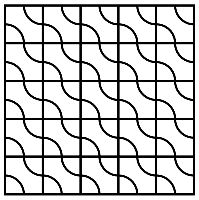
---

[34] For 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].

```

\newcount\Boolean

\def\BasicTileLPT{% LPT tile
\setrannum{\Boolean}{0}{1}% From random.tex (Donald Arseneau)
\ifnum\Boolean=0
  \ProtoTileLPT%
\else
  \rotateleft{\ProtoTileLPT}%
\fi}

\ProtoTileLPT\hfill
\rotateleft{\ProtoTileLPT}\hfill
\psset{unit=0.5}%
\psboxfill{\BasicTileLPT}
\Tiling{}{(5,5)}
```



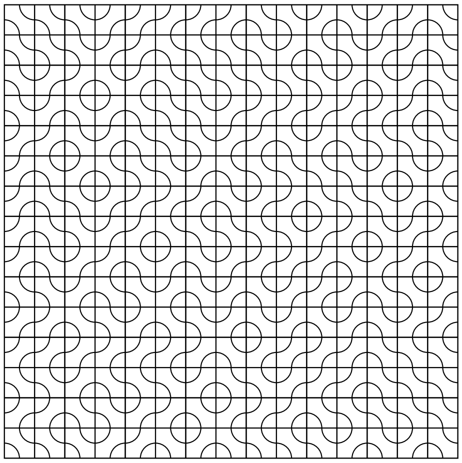




For simple cases, there is a solution to this problem using a mixture of PSTricks and PostScript programming. Here the PSTricks construction `\pscustom{\code{...}}` allows us to insert PostScript code inside the LaTeX+ PSTricks one. The programming is less straightforward than to solve this problem using 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 TeX memory. Note also that **\pslbrace** and **\psrbrace** are PSTricks macros which insert the { and } characters.

```
\def\ProtoTileLPT{{%
\psset{dimen=middle}%
\psframe(1,1)
\psarc(0,0){0.5}{0}{90}
\psarc(1,1){0.5}{-180}{-90}}}

\newcount\InitCounter% Counter to change the random seed

\def\BasicTileLPT{% LPT tile
\InitCounter=\the\time
\pscustom{\code{rand \the\InitCounter\space sub 2 mod
  0 eq \pslbrace}}
\pspicture(1,1)\ProtoTileLPT\endpspicture%
\pscustom{\code{\psrbrace \pslbrace}}
\rotateleft{\ProtoTileLPT}%
\pscustom{\code{\psrbrace ifelse}}}

```

```
\psset{unit=0.4,linewidth=0.4pt}
\psboxfill{\BasicTileLPT}
\Tiling{}{(15,15)}
```

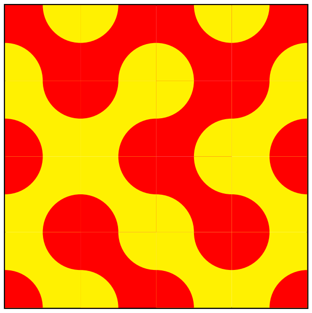


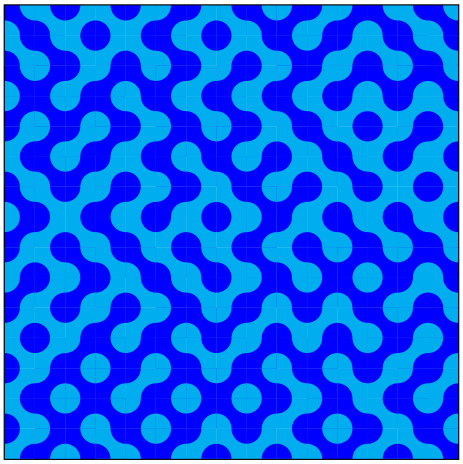
Using the very surprising fact that the coloring of these tiles does not depend on their neighbors (even if it is difficult to believe as the opposite seems obvious!) but only on the parity of the value of row and column positions, we can directly program in the same way a colored version of the LPT 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.

```
\def\ProtoTileLPT#1#2{% LPT prototile
\psset{dimen=middle,linestyle=none,fillstyle=solid}%
\psframe[fillcolor=#1](1,1)
\psset{fillcolor=#2}%
\pswedge(0,0){0.5}{0}{90}
\pswedge(1,1){0.5}{-180}{-90}}

\newcount\InitCounter% Counter to change the random seed

\def\BasicTileLPT#1#2{% LPT tile
\InitCounter=\the\time
\pscustom{\code{rand \the\InitCounter\space sub 2 mod
   0 eq \pslbrace row column add 2 mod 0 eq \pslbrace}}
\pspicture(1,1)\ProtoTileLPT{#1}{#2}\endpspicture%
\pscustom{\code{\psrbrace \pslbrace}}
\ProtoTileLPT{#2}{#1}%
\pscustom{\code{\psrbrace ifelse \psrbrace \pslbrace
   row column add 2 mod 0 eq \pslbrace}}
\rotateleft{\ProtoTileLPT{#2}{#1}}%
\pscustom{\code{\psrbrace \pslbrace}}
\rotateleft{\ProtoTileLPT{#1}{#2}}%
\pscustom{\code{\psrbrace ifelse \psrbrace ifelse}}}

```

```
\psboxfill{\BasicTileLPT{red}{yellow}}
\Tiling{}{(4,4)}
\hfill
\psset{unit=0.4}
\psboxfill{\BasicTileLPT{blue}{cyan}}
\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.

```
% \Escape will be the \ character
{\catcode‘\!=0\catcode‘\\=11!gdef!Escape{\}}

\def\ProtoTile{%
\pspicture(1,1)\psframe[dimen=middle](1,1)\endpspicture%
\pscustom{%
  \moveto(-0.9,0.75)% In PSTricks units
  \code{/Times-Italic findfont 8 scalefont setfont
    (\Escape() show row 3 string cvs show (,) show
    column 3 string cvs show (\Escape)) show}
  \moveto(-0.5,0.25)% In PSTricks units
  \code{/Times-Bold findfont 18 scalefont setfont
    1 0 0 setrgbcolor% red color
    /center {dup stringwidth pop 2 div neg 0 rmoveto} def
    row 1 sub x2 mul column add 3 string cvs center show}}}
\psboxfill{\ProtoTile}
\Tiling{}{(6,4)}
```

(1,1) **1**	*(1,2)* **2**	*(1,3)* **3**	*(1,4)* **4**	*(1,5)* **5**	*(1,6)* **6**
(2,1) **7**	*(2,2)* **8**	*(2,3)* **9**	*(2,4)* **10**	*(2,5)* **11**	*(2,6)* **12**
(3,1) **13**	*(3,2)* **14**	*(3,3)* **15**	*(3,4)* **16**	*(3,5)* **17**	*(3,6)* **18**
(4,1) **19**	*(4,2)* **20**	*(4,3)* **21**	*(4,4)* **22**	*(4,5)* **23**	*(4,6)* **24**

X Three Dimensional Graphics[36]

51 Overview

pst-3d

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

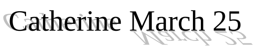
\psshadow[*par*]**{*stuff*}**

is a macro to draw a text with a shadow.

```
\psshadow{\Huge  Shadow}
```

The **Tshadowangle** parameter specifies the angle of the shadow in degrees (do not use a value of 0 or 180).

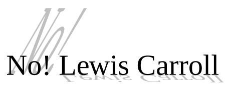
Tshadowangle=*num* **Default: 60**



```
\psshadow[Tshadowangle=120]{Catherine}

\psshadow[Tshadowangle=-45]{March  25}
```

The **Tshadowsize** parameter specifies the size of the shadow.

Tshadowsize=*dim* **Default: 1**



```
\psshadow[Tshadowsize=4]{No!}

\psshadow[Tshadowangle=-45,Tshadowsize=0.5]{Lewis  Carroll}
```

---

[36] This chapter is merely an adaptation of the documentation for this package written in 1998 by Manuel Luque.

[37] For 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**

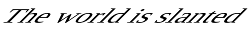
Good morning!
Good morning!

```
\psshadow[Tshadowsize=2,Tshadowcolor=red]{Good  morning!}
```

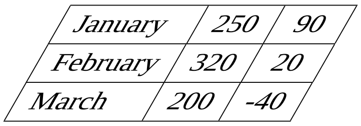
Note that the result will be incorrect if you change the color of the text itself from black (then the shadow will be the same color as the text).

\pstilt[*par*]{*angle*}{*stuff*}

is a macro which angles an object into the third dimension (do not use an angle of 0 or 180).



```
\pstilt{45}{The  world  is  slanted}
```



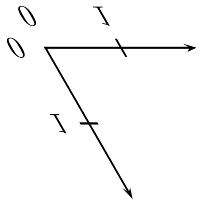
```
\pstilt{60}{%
   \begin{tabular}{|l|c|c|}
      \hline
      January   &  250  &   90  \\\hline
      February  &  320  &   20  \\\hline
      March     &  200  &  -40  \\\hline
   \end{tabular}}
```

\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

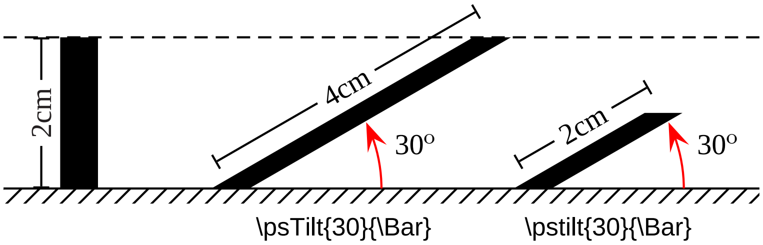
```
\psTilt{45}{The  world  is  slanted}
```



```
\psTilt{-60}{%
   \begin{pspicture}(2,2)
      \psaxes{<->}(2,2)
   \end{pspicture}}
```

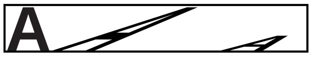
The difference between **\pstilt** and **\psTilt** is that **\pstilt** keeps the length of the objects constant while **\psTilt** keeps the height of the objects constant as shown in the following example:

```
\def\Bar{\psframe*(-0.25,0)(0.25,2)}

% Additional information is superimposed on the drawing
\rput(1,0){\Bar}
\rput(3,0){\psTilt{30}{\Bar}}
\rput(7,0){\pstilt{30}{\Bar}}
```



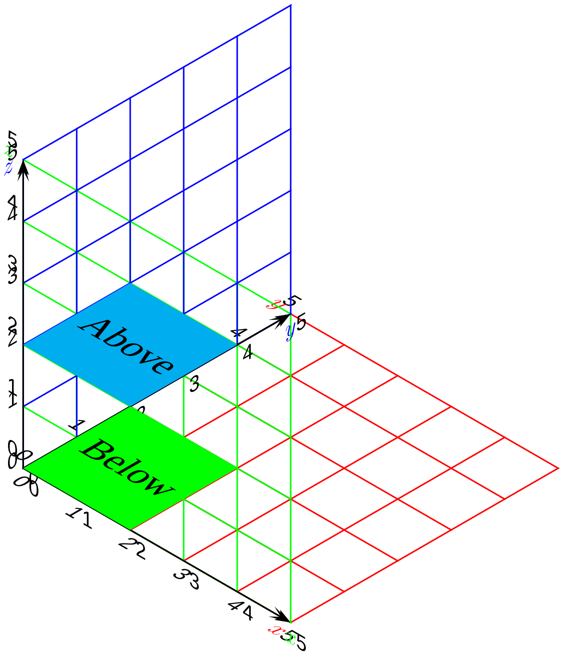
These two effects can now easily be understood:

```
\psframebox[framesep=0]{\Huge\sffamily\bfseries%
  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).



```
% The three colored grids are previously drawn
\ThreeDput{\psframe*[linecolor=green](2,2)}
\ThreeDput(1,1,0){\LARGE Below}
\ThreeDput(0,0,2){\psframe*[linecolor=cyan](2,2)}
\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

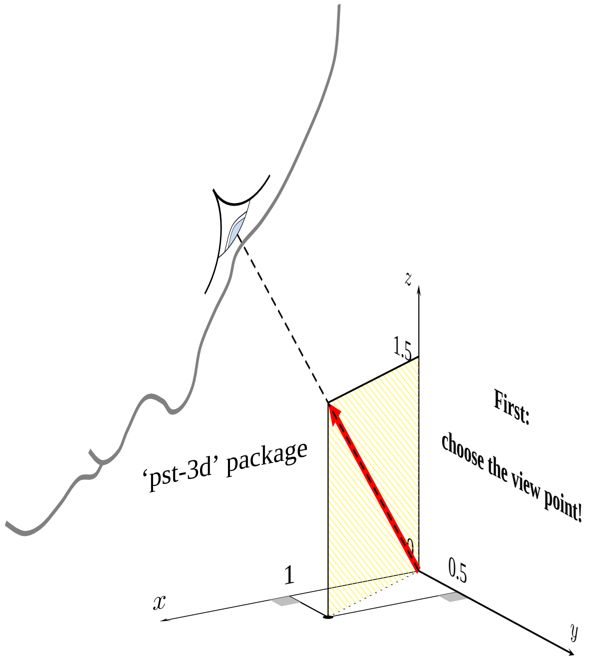


Figure 2: How to choose the viewpoint for the **\ThreeDput** macro.

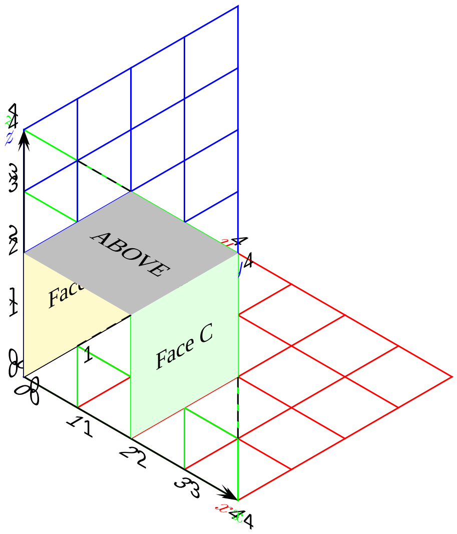
on the horizontal plane. As the vector norm has no importance, it is not necessary to take a unit vector. We must simply choose the coordinates according to the view point that we must have on the object. In figure 2, this vector has *(1 0.5 1.5)* for coordinates.[38]

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.

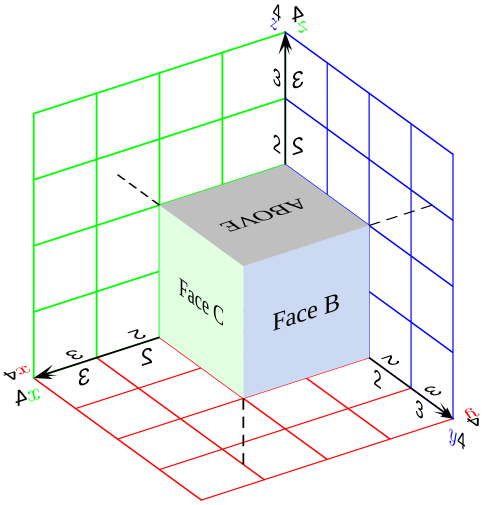
The cube shown next has its faces drawn in the following order: *Face A* (yellow), *Face B* (blue), *Face C* (green) then face marked *ABOVE* (gray). The various objects are drawn one after the other and there is no management of hidden-line removal. You can see that the order used is very important and must sometimes be changed according to the viewpoint chosen (see the fourth of the following examples).

---

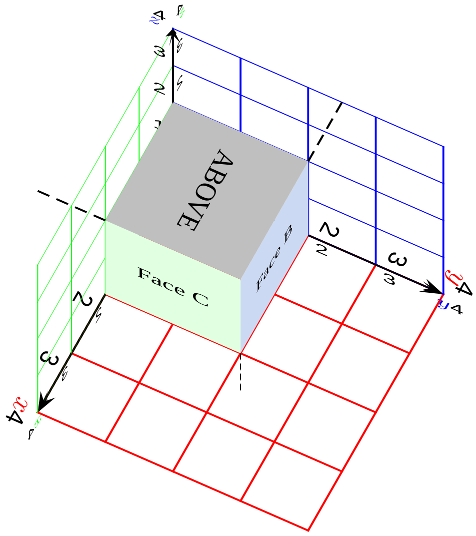
[38] But the figure itself is not drawn with this value, because we would only see a single point, at the end of the arrow!



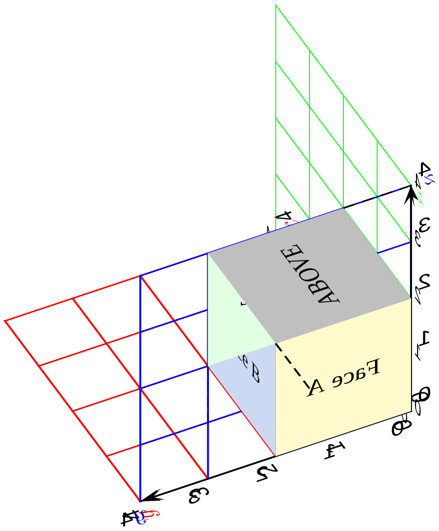
```
\psset{viewpoint=1 -1 1}% Default value
\ShowPartialCube
% Note that Face A on the left side is partially covered
% and that Face B is totally covered
```



```
\psset{viewpoint=1 1.5 1}
\ShowPartialCube
```



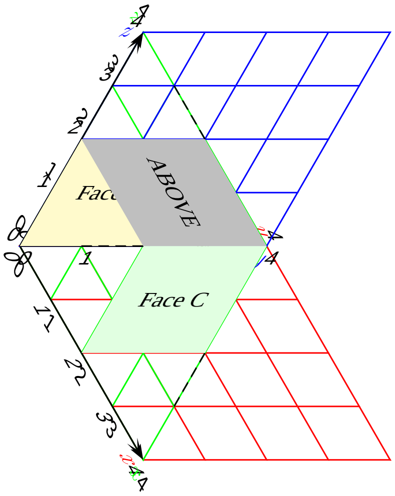
```
\psset{viewpoint=1 0.5 2}
\ShowPartialCube
```



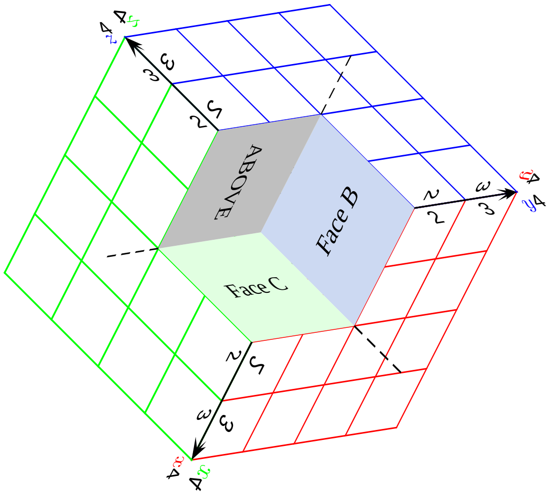
```
\psset{viewpoint=-1  0.5  1}
\ShowPartialCube
% Note that as the order of drawing is faces A, B and C.
% C partially covers the previously drawn B.
% This illustrates the importance of the order in which the
% different objects of a three dimensional scene are drawn.
```

The **viewangle** parameter specifies the clockwise rotation applied from the view point.

viewangle=*angle* **Default: 0**



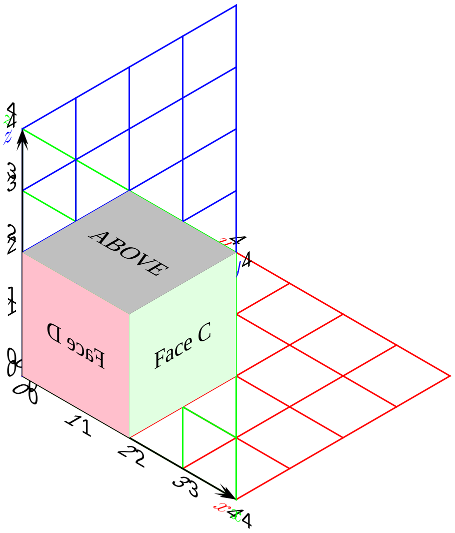
```
\psset{viewangle=30}
\ShowPartialCube
```



```
\psset{viewpoint=1  1.5  1,viewangle=-45}
\ShowPartialCube
```

The **normal** parameter specifies the normal vector.

normal=n_x n_y n_z **Default: 0 0 1**



```
\def\Face#1#2{\psframe*[linecolor=#1](2,2)\rput(1,1){#2}}

% The three colored grids are previously drawn
\ThreeDput[normal=0 1 0](2,0,0){\Face{Pink}{Face D}}
\ThreeDput[normal=1 0 0](2,0,0){\Face{PaleGreen}{Face C}}
\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: $\left| \begin{array}{lcl} n_x & = & 1 \\ n_y & = & 0 \\ n_z & = & 0 \end{array} \right.$

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.

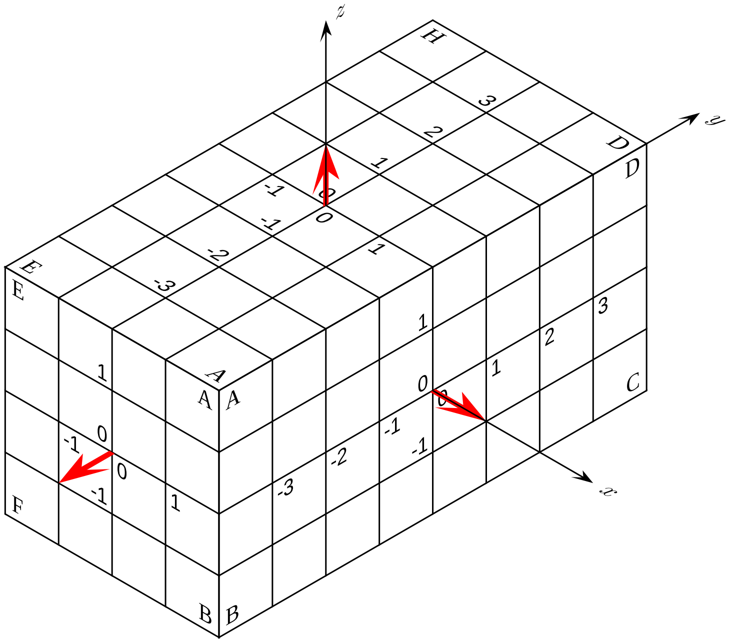
The normal to the face ADHE has the coordinates: $\left| \begin{array}{lcl} n_x & = & 0 \\ n_y & = & 0 \\ n_z & = & 1 \end{array} \right.$

The normal to the face AEFB has the coordinates: $\left| \begin{array}{lcr} n_x & = & 0 \\ n_y & = & -1 \\ n_z & = & 0 \end{array} \right.$

etc.

```
% The three visible faces of the rectangle parallelopiped
% (note that each time we will call the grid macro twice
% once for the grid itself and secondly for the labels,
% because we do not want to label the extreme values)

\def\FaceABCD{%
   \psgrid[gridlabels=0](0,0)(-4,-2)(4,2)
   \psgrid[gridwidth=0](0,0)(-3,-1)(3,1)
   \uput[-45](-4,2){A}
   \uput[45](-4,-2){B}
   \uput[135](4,-2){C}
   \uput[225](4,2){D}}

\def\FaceADHE{%
   \psgrid[gridlabels=0](0,0)(-2,-4)(2,4)
   \psgrid[gridwidth=0](0,0)(-1,-3)(1,3)
   \uput[135](2,-4){A}
   \uput[225](2,4){D}
   \uput[-45](-2,4){H}
   \uput[45](-2,-4){E}}

\def\FaceABFE{%
   \psgrid[gridlabels=0](0,0)(-2,-2)(2,2)
   \psgrid[gridwidth=0](0,0)(-1,-1)(1,1)
   \uput[225](2,2){A}
   \uput[135](2,-2){B}
   \uput[45](-2,-2){F}
   \uput[-45](-2,2){E}}

\pspicture(-4.2,-4.1)(5.3,4.3)
   \psset{dimen=middle,subgriddiv=0,arrows=->,arrowscale=2}%
   %
   % Face ABCD
   \ThreeDput[normal=1 0 0](2,0,0){\FaceABCD}
   %
   % Normal for the plane ABCD.
   % In its center, we draw the plane where it is placed.
   % This is the plane parallel to Oxy placed in (2,0,0)
   % for which the normal is parallel to Oz.
   \ThreeDput[normal=0 0 1](2,0,0){%
      \psline[linecolor=red,linewidth=0.1](1,0)
      \psline(3,0)\uput[0](3,0){$x$}}
   %
   % Face ADHE
   \ThreeDput[normal=0 0 1](0,0,2){\FaceADHE}
   %
   % Normal for the plane ADHE.
   % In its center, we draw the plane where it is placed.
   % This is the plane parallel to Oyz placed in (0,0,2)
   % for which the normal is parallel to Ox.
   \ThreeDput[normal=1 0 0](0,0,2){%
      \psline[linecolor=red,linewidth=0.1](0,1)
      \psline(0,3)\uput[0](0,3){$z$}}
   %
   % Face ABFE
   \ThreeDput[normal=0 -1 0](0,-4,0){\FaceABFE}
   %
   % Normal for the plane ABFE.
   % In its center, we draw the plane where it is placed.
   % This is the plane parallel to Oxy placed in (0,-4,0)
   % for which the normal is parallel to Oy.
   \ThreeDput[normal=0 0 -1](0,-4,0){%
      \psline[linecolor=red,linewidth=0.1](0,1)}
   %
```

```
  % Oy axis
  \ThreeDput[normal=0 0 1](0,4,0){%
     \psline(0,3)\uput[0](0,3){$y$}}
\endpspicture
```

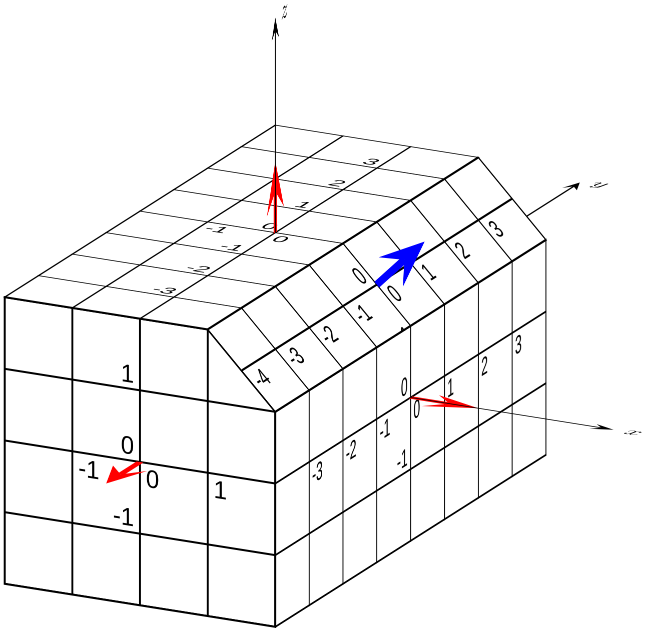


Let us suppose that we wish to draw a chamfered edge according to the edge AD. The normal for the chamfer has the coordinates: $\left|\begin{array}{lcl} n_x & = & 1\\ n_y & = & 0\\ n_z & = & 1\end{array}\right.$ and the origin of this plane has the coordinates: $\left|\begin{array}{lcr} x_0 & = & 1.5\\ y_0 & = & 0\\ z_0 & = & 1.5\end{array}\right.$

The dimensions of this chamfered edge are $L = 8, l = \sqrt{2} = 1.414$. We will draw the chamfer with its normal.

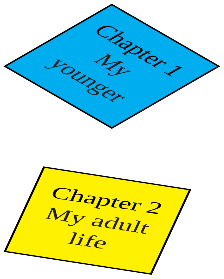
```
\pspicture(-3.6,-3.4)(4.8,4.9)
\psset{dimen=middle,subgriddiv=0,arrows=->,arrowscale=2,
        viewpoint=1 -2 0.75}%
%
% Oy axis
\ThreeDput[normal=0 0 1](0,4,0){%
  \psline(0,5)\uput[0](0,5){$y$}}
%
\ThreeDput[normal=1 0 0](2,0,0){%
  \psgrid[gridlabels=0](0,0)(-4,-2)(4,1)
  \psgrid[gridwidth=0](0,0)(-3,-1)(3,0)}
%
\ThreeDput[normal=0 0 1](2,0,0){%
  \psline[linecolor=red,linewidth=0.1](1,0)
  \psline(3,0)\uput[0](3,0){$x$}}
%
\ThreeDput[normal=0 0 1](0,0,2){%
  \psgrid[gridlabels=0](0,0)(-2,-4)(1,4)
  \psgrid[gridwidth=0](0,0)(-1,-3)(1,3)}
```

```
%
\ThreeDput[normal=1 0 0](0,0,2){%
   \psline[linecolor=red,linewidth=0.1](0,1)
   \psline(0,3)\uput[0](0,3){$z$}}
%
\ThreeDput[normal=0 -1 0](0,-4,0){%
   \psclip{\pspolygon(-2,-2)(-2,2)(1,2)(2,1)(2,-2)}
      \psgrid[gridlabels=0](0,0)(-2,-2)(2,2)
      \psgrid[gridwidth=0](0,0)(-1,-1)(1,1)
   \endpsclip}
%
\ThreeDput[normal=0 0 -1](0,-4,0){%
   \psline[linecolor=red,linewidth=0.1](0,1)}
%
% The chamfered edge
\ThreeDput[normal=1 0 1](1.5,0,1.5){%
   \psclip{\psframe[fillstyle=solid](-4,-0.707)(4,0.707)}
      \psgrid(0,0)(-4,-0.707)(4,0.707)
   \endpsclip}
%
% The normal vector to this chamfer
\ThreeDput[normal=0 1 0](1.5,0,1.5){%
   \psline[linecolor=blue,linewidth=0.1](-0.707,0.707)}
\endpspicture
```

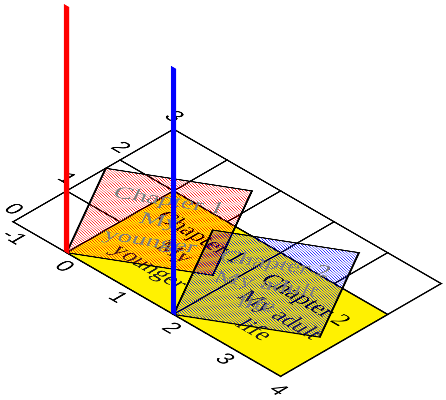


The **embedangle** parameter rotates the object about the normal in a counter clockwise (if embedangle > 0) direction.

embedangle=*angle* **Default: 0**



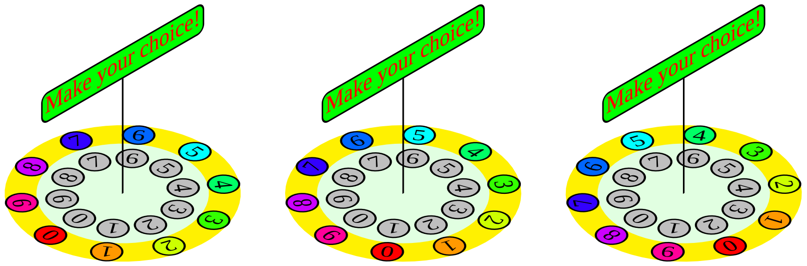
```
\def\BookChapter#1#2#3#4{%
\psframe[style=#4](2,2)
\rput(1,1){%
   \begin{minipage}{1.5cm}
      \centering
      \textcolor{#3}{Chapter #1}\\
      #2
   \end{minipage}}}

\newpsstyle{SolidCyan}{fillstyle=solid,fillcolor=cyan}
\newpsstyle{SolidYellow}{fillstyle=solid,fillcolor=yellow}

\ThreeDput(-2,-2,0){%
   \BookChapter{1}{My younger}{black}{SolidCyan}}

\ThreeDput[embedangle=30](-2,-2,-3){%
   \BookChapter{2}{My adult life}{black}{SolidYellow}}
```



```
\newpsstyle{TransparencyRed}{fillstyle=vlines,hatchcolor=red,
   hatchwidth=0.1\pslinewidth,hatchsep=1\pslinewidth}
\newpsstyle{TransparencyBlue}{fillstyle=vlines,hatchcolor=blue,
   hatchwidth=0.1\pslinewidth,hatchsep=1\pslinewidth}

% Underlaying grid
\ThreeDput{\psgrid[subgriddiv=0](-1,0)(4,3)}

% embedangle=0
\ThreeDput{%
   \BookChapter{1}{My younger}{black}{SolidYellow}}
\ThreeDput(2,0,0){%
   \BookChapter{2}{My adult life}{black}{SolidYellow}}

\bgroup
   \psset{embedangle=30}%
   % embedangle=30
   \ThreeDput{%
      \BookChapter{1}{\textcolor{gray}{My younger}}
                     {gray}{TransparencyRed}}
   \ThreeDput(2,0,0){%
      \BookChapter{2}{\textcolor{gray}{My adult life}}
                     {gray}{TransparencyBlue}}
\egroup

% Normals
\ThreeDput[normal=0 1 0]{%
   \psline[linewidth=0.1,linecolor=red](0,4)}
\ThreeDput[normal=0 1 0](2,0,0){%
   \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.

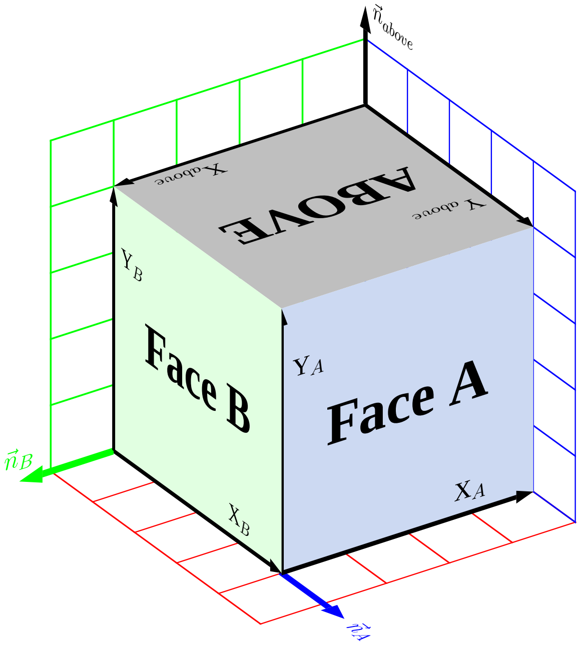
```
\psset{unit=0.3,dimen=middle,framesep=0.15,viewpoint=1 1 1}
\SpecialCoor

\multido{\iEmbedAngle=0+36}{3}{%
  \pspicture(-5.2,-3)(5.2,8.3)
    \ThreeDput{%
      \pscircle*[linecolor=PaleGreen]{4.5}
      \multido{\iNum=0+1,\iAngle=0+36,\iRot=-90+36}{10}{%
        \rput{\iRot}(2.7;\iAngle){%
         \pscirclebox[fillstyle=solid,fillcolor=lightgray]{%
           \small\iNum}}}}
    \ThreeDput[embedangle=\iEmbedAngle]{%
      \pscircle[linewidth=1.4,linecolor=yellow]{4.5}
      \multido{\iNum=0+1,\iAngle=0+36,\iRot=-90+36,
             \nHue=0.0+0.1}{10}{%
        \definecolor{ColorNumber}{hsb}{\nHue,1,1}%
        \rput{\iRot}(4.5;\iAngle){%
          \pscirclebox[fillstyle=solid,fillcolor=ColorNumber]{%
            \small\iNum}}}}
    \ThreeDput[normal=0 1 0]{%
      \psline[linewidth=0.1](0,7)
      \rput(0,7){%
        \psframebox[fillstyle=solid,fillcolor=green,framearc=.5]{%
          \textcolor{red}{Make your choice!}}}}
  \endpspicture
  \hfill}
```



53 More details

The macro **\ThreeDput**[normal=n$_x$ n$_y$ n$_z$](Ω_x Ω_y Ω_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.



```
\psset{viewpoint=1  1.5  1,arrows=->,arrowinset=0,
        gridlabels=0,subgriddiv=0}
%  Grids
\ThreeDput[normal=0  0  1]{%  Oxy  plane
   \psgrid[gridcolor=red](5,5)
   \psline[linewidth=3pt,linecolor=blue](4,4)(4,5.5)
   \uput[90](4,5.5){%
      \rotateleft{\textcolor{blue}{$\vec{n}_A$}}}}
\ThreeDput[normal=0  -1  0]{%  Oxz  plane
   \psgrid[gridcolor=green](5,5)
   \psline[linewidth=3pt,linecolor=green](4,0)(5.5,0)
   \uput[90](5.5,0){%
      \scalebox{-1  1}{\textcolor{green}{$\vec{n}_B$}}}}
\ThreeDput[normal=1  0  0]{%  Oyz  plane
   \psgrid[gridcolor=blue](5,5)
   \psline[linewidth=3pt](0,4)(0,5.5)
   \uput[0](0,5.5){$\vec{n}_{above}$}}
%  Cube  and  axes
\psset{linewidth=2pt}
\ThreeDput[normal=0  0  1](0,0,4){%
   \psframe*[linecolor=lightgray](4,4)
      \rput(2,2){\Huge\textbf{ABOVE}}
   \psline(4,0)\uput[90](3,0){X$_{above}$}
   \psline(0,4)\uput[0](0,3){Y$_{above}$}}
\ThreeDput[normal=0  1  0](4,4,0){%
   \psframe*[linecolor=LightBlue](4,4)
      \rput(2,2){\Huge\textbf{Face  A}}
   \psline(4,0)\uput[90](3,0){X$_{A}$}
   \psline(0,4)\uput[0](0,3){Y$_{A}$}}
\ThreeDput[normal=1  0  0](4,0,0){%
   \psframe*[linecolor=PaleGreen](4,4)
      \rput(2,2){\Huge\textbf{Face  B}}
   \psline(4,0)\uput[90](3,0){X$_{B}$}
   \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 $\Omega\,\mathrm{X}_\mathrm{B}$,$\Omega\,\mathrm{Y}_\mathrm{B}$,$\vec{n}_\mathrm{B}$ produces a direct trihedral. From a practical point of view, we must have (the three fingers of your right hand producing a trihedral):

- thumb = $\Omega\,\mathrm{X}_\mathrm{B}$
- forefinger = $\Omega\,\mathrm{Y}_\mathrm{B}$
- long finger = $\vec{n}_\mathrm{B}$

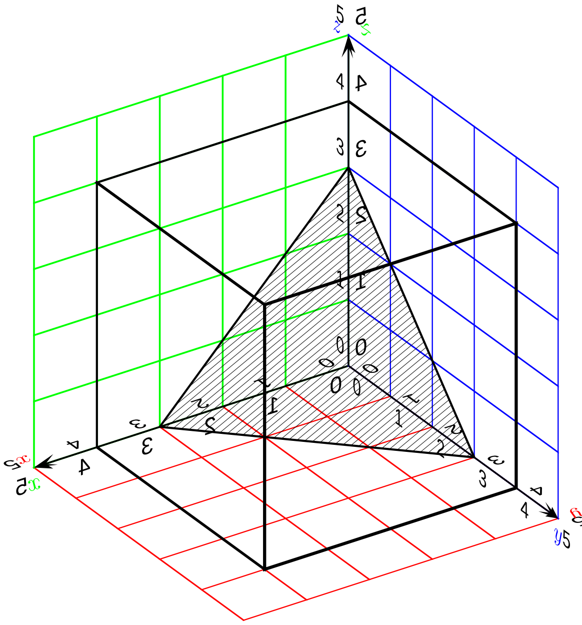
All the faces obey to this rule. Imagine standing on a face of the cube with the heels of your feet meeting at one corner, your right foot will be facing Ox, your left foot will be facing Oy and you will be standing in the $\vec{n}$ (the normal) direction. You can write onto the face in the direction that your right foot points toward.

How to represent the intersection of the unit cube with the plane h defined by:

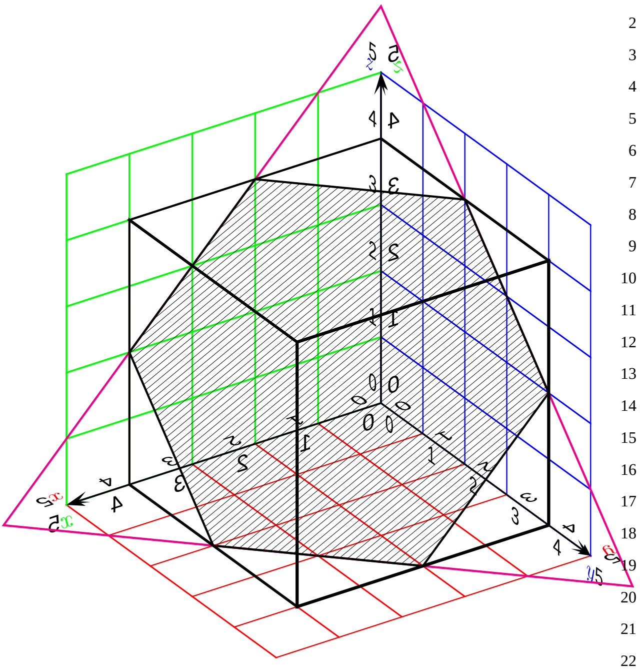
- $0\leq x\leq 4$
- $0\leq y\leq 4$
- $0\leq z\leq 4$

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 $\Omega(\frac{\mathtt{h}}{3}, \frac{\mathtt{h}}{3}, \frac{\mathtt{h}}{3})$ 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]$(\frac{h}{3}, \frac{h}{3}, \frac{h}{3})$

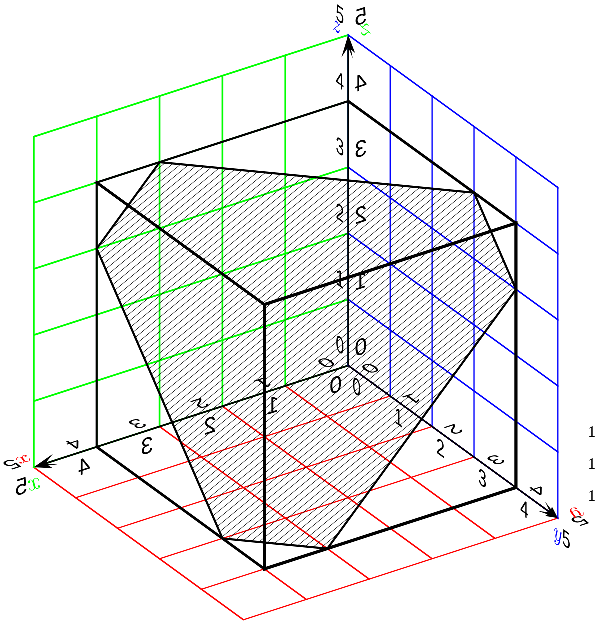
The next examples are related respectively to h=3, h=6 (regular hexagon) and h=7. But, first, we must compute the coordinates of the vertex of the polygons (triangle or hexagon), intersections of the plane with the edges of the cube (this package does not do it, but we can easily code the computations in PostScript). After that we can draw the polygon by the usual **\pspolygon** macro.



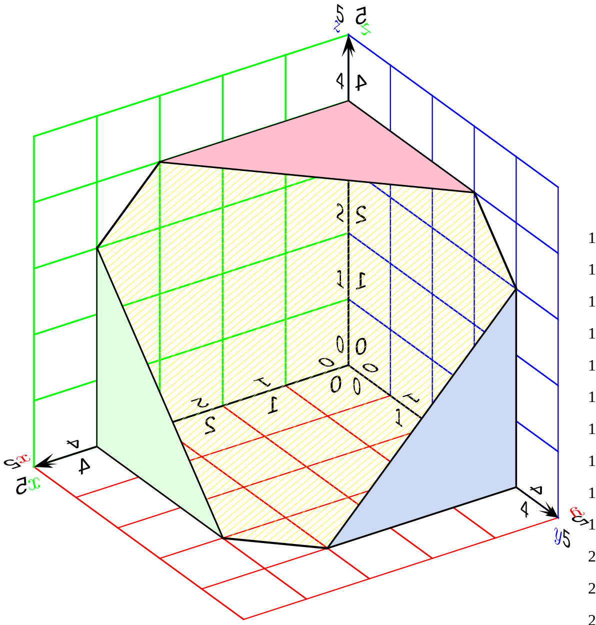
```
\SpecialCoor
\psset{viewpoint=1 1.5 1,arrows=->,arrowscale=2,subgriddiv=0}
% Grids and axes
\ThreeDput[normal=0 0 1]{% Oxy plane
  \psgrid[gridcolor=red](5,5)
  \psline(0,5)\uput[180](0,5){\textcolor{red}{$y$}}
  \psline(5,0)\uput[-90](5,0){\textcolor{red}{$x$}}}
\ThreeDput[normal=0 -1 0]{% Oxz plane
  \psgrid[gridcolor=green](5,5)
  \psline(0,5)\uput[180](0,5){\textcolor{green}{$z$}}
  \psline(5,0)\uput[-90](5,0){\textcolor{green}{$x$}}}
\ThreeDput[normal=1 0 0](0,0,0){% Oyz plane
  \psgrid[gridcolor=blue](5,5)
  \psline(0,5)\uput[180](0,5){\textcolor{blue}{$z$}}
  \psline(5,0)\uput[-90](5,0){\textcolor{blue}{$y$}}}
% Intersection of the plane x+y+z=h with the cube for h=3
% Radius = h*sqrt(2/3) = 3*0.8465 = 2.4495
\ThreeDput[normal=1 1 1](1,1,1){% (h/3,h/3,h/3)
  \pscustom[fillstyle=hlines,hatchwidth=0.1pt,hatchsep=2pt]{%
   \code{/h 3 def /Radius h 2 3 div sqrt mul def}%
   \pspolygon(! Radius dup  -30 cos mul exch -30 sin mul)
             (! Radius dup  90 cos mul exch  90 sin mul)
             (! Radius dup 210 cos mul exch 210 sin mul)}}
% Cube
\psset{linewidth=0.05}
\ThreeDput[normal=0 0 1](0,0,4){\psframe(4,4)}
\ThreeDput[normal=0 1 0](4,4,0){\psframe(4,4)}
\ThreeDput[normal=1 0 0](4,0,0){\psframe(4,4)}
```



```
\SpecialCoor
\psset{viewpoint=1 1.5 1,arrows=->,arrowscale=2,subgriddiv=0}
% Grids and axes
% ...
% Polygons
\ThreeDput[normal=1 1 1](2,2,2){% (h/3,h/3,h/3)
  % Triangle
  \pscustom[linecolor=magenta]{%
   \code{/h 6 def /Radius h 2 3 div sqrt mul def}%
   \pspolygon(! Radius dup  -30 cos mul exch -30 sin mul)
             (! Radius dup  90 cos mul exch  90 sin mul)
             (! Radius dup 210 cos mul exch 210 sin mul)}
  % Intersection of the plane x+y+z=h with the cube for h=6
  % Radius = h/3*sqrt(2) = 6/3*1.414 = 2.828
  \pscustom[fillstyle=hlines,hatchwidth=0.1pt,hatchsep=2pt]{%
   \code{/h 6 def /Radius h 3 div 2 sqrt mul def}%
   \pspolygon(! Radius dup   0 cos mul exch   0 sin mul)
             (! Radius dup  60 cos mul exch  60 sin mul)
             (! Radius dup 120 cos mul exch 120 sin mul)
             (! Radius dup 180 cos mul exch 180 sin mul)
             (! Radius dup 240 cos mul exch 240 sin mul)
             (! Radius dup 300 cos mul exch 300 sin mul)}}
% Cube
% ...
```



```
\SpecialCoor
\psset{viewpoint=1 1.5 1,arrows=->,arrowscale=2,subgriddiv=0}
% Grids and axes
% ...
% Intersection of the plane x+y+z=h with the cube for h=7
\def\Radius{2.9439}%
\ThreeDput[normal=1 1 1](2.33,2.33,2.33){% (h/3,h/3,h/3)
  \pspolygon[fillstyle=hlines,hatchwidth=0.1pt,hatchsep=2pt]
    (\Radius;43.9)(\Radius;136.1)(\Radius;163.9)
    (\Radius;256.1)(\Radius;283.9)(\Radius;376.1)}
% Cube
% ...
```



```
\SpecialCoor
\psset{viewpoint=1 1.5 1,arrows=->,arrowscale=2,subgriddiv=0}
% Grids and axes
% ...
% Intersection of the plane x+y+z=h with the cube for h=7
\def\Radius{2.9439}%
\ThreeDput[normal=1 1 1](2.33,2.33,2.33){% (h/3,h/3,h/3)
  \pspolygon[fillstyle=hlines,hatchwidth=0.1pt,hatchsep=2pt,
             hatchcolor=yellow]
    (\Radius;43.9)(\Radius;136.1)(\Radius;163.9)
    (\Radius;256.1)(\Radius;283.9)(\Radius;376.1)}
%
% On the face A, this is a triangle with edges
% of coordinates (4,0)(1,0)(4,3)
\ThreeDput[normal=0 1 0](4,4,0){%
  \pspolygon[fillstyle=solid,fillcolor=LightBlue](4,0)(1,0)(4,3)}
% On the face B, this is a triangle with edges
% of coordinates (0,0)(0,3)(3,0)
\ThreeDput[normal=1 0 0](4,0,0){%
  \pspolygon[fillstyle=solid,fillcolor=PaleGreen](0,0)(0,3)(3,0)}
% On the upper face, this is a triangle with edges
% of coordinates (0,0)(3,0)(0,3)
\ThreeDput[normal=0 0 1](0,0,4){%
  \pspolygon[fillstyle=solid,fillcolor=Pink](0,0)(3,0)(0,3)}
```

XI

Other features

54 Special coordinates

The command

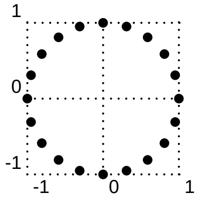
\SpecialCoor

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:

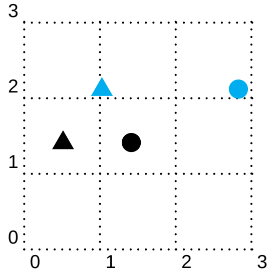
(*x*,*y*) The usual Cartesian coordinate. E.g., (3,4).

(*r*;*a*) Polar coordinate, with radius r and angle a. The default unit for r is **unit**. E.g., (3;110).



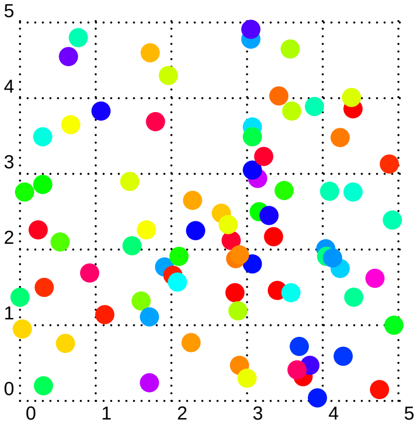
```
\SpecialCoor
\multido{\iAngle=0+18}{20}{\psdot(1;\iAngle)}
```

(!*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:



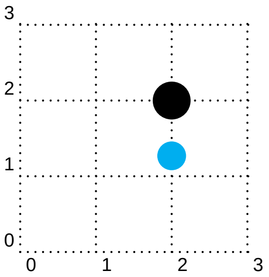
```
\SpecialCoor
\psset{dotscale=2,xunit=2,yunit=1.5}%
\psdot(2;45)
\psdot[linecolor=cyan](! 2 45 cos mul 2 45 sin mul)
\psset{dotstyle=triangle*}%
\psdot(1.5;70)
\psdot[linecolor=cyan](! 1.5 70 cos mul 1.5 70 sin mul)
```

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]



```
\SpecialCoor
\psset{dotscale=2}%
\multips(0,0){80}{%
   \psdot(! rand 501 mod 100 div rand 501 mod 100 div
             rand 101 mod 100 div 1 1 sethsbcolor)}
```

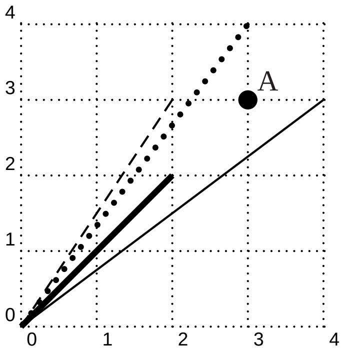
(*coor1*|*coor2*) The x coordinate from *coor1* and the y coordinate from *coor2*. *coor1* and *coor2* can be any other coordinates for use with **\SpecialCoor**.



```
\SpecialCoor
\Cnode*(2,2){A}
\psdot[linecolor=cyan,dotscale=3](A|1in;30)
```

(*node*) The center of *node*. E.g., (A).

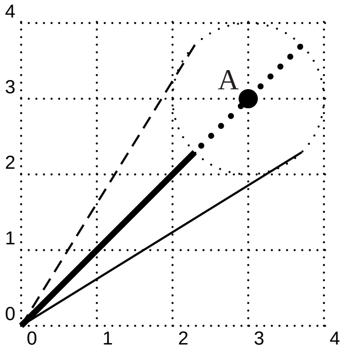
([*par*]*node*) The position relative to *node* determined using the **angle**, **nodesep** and **offset** optional parameters. E.g., ([angle=45]A). Using both **nodesep** and **offset** allow to define relative moves (**nodesep** for horizontal and **offset** for vertical) from an existing node.



```
\SpecialCoor
\pnode(3,3){A}\psdot[dotscale=2](A)\uput[45](A){A}
\psline([nodesep=1]A)
\psline[linestyle=dashed]([nodesep=-1]A)
\psline[linestyle=dotted,linewidth=0.08]([offset=1]A)
\psline[linewidth=0.08]([nodesep=-1,offset=-1]A)
```

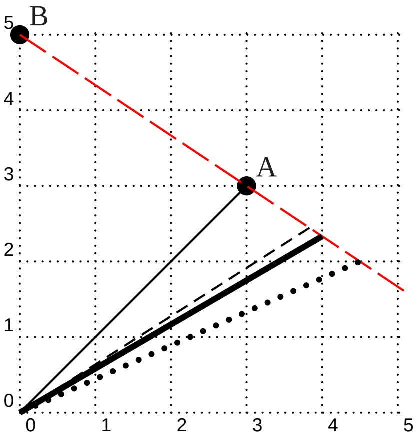
---

[39] The 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.)



```
\SpecialCoor
\pnode(3,3){A}\psdot[dotscale=2](A)\uput[135](A){A}
\pscircle[linestyle=dotted](A){1}
\psline([nodesep=1,angle=-45]A)
\psline[linestyle=dashed]([nodesep=-1,angle=-45]A)
\psline[linestyle=dotted,linewidth=0.08]([offset=1,angle=-45]A)
\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.

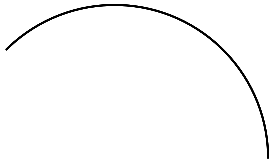


```
\SpecialCoor
\pnode(3,3){A}\psdot[dotscale=2](A)\uput[45](A){A}
\pnode(0,5){B}\psdot[dotscale=2](B)\uput[45](B){B}
\psline[linestyle=dashed,dash=0.4  0.1,linecolor=red]
        (B)([nodesep=-2.5]{B}A)
\psline(A)
\psline[linestyle=dashed]([nodesep=-1]{B}A)
\psline[linewidth=0.08]([Ynodesep=-1]{B}A)
\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:



```
\SpecialCoor
\psarc(0,0){.8in}{0}{135}
\psarc(0,0){.8in}{0}{(-1,1)}
```

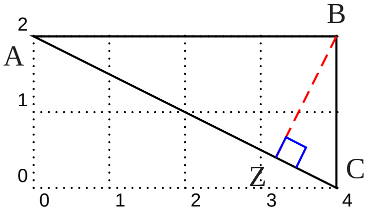
!*ps* Raw PostScript code. *ps* should expand to a number. The same units are used as with *num*.

The command

\NormalCoor

disables the **\SpecialCoor** features.

Here is an example for geometric drawing, which use some PostScript computations and node references.[40]



```
\SpecialCoor

\pnode(0,2){A}\uput[-135](A){A}
\pnode(4,2){B}\uput[90](B){B}
\pnode(4,0){C}\uput[45](C){C}
\pspolygon(A)(B)(C)
% Angle = atan(2/4) = 26.57
\pnode(! /LengthA 4 def
         /LengthB 2 def
         /Angle LengthB LengthA atan def
         LengthA Angle cos dup mul mul
         LengthB Angle cos Angle sin LengthA mul mul
           sub){Z}\uput[-135](Z){Z}
\psline[linecolor=red,linestyle=dashed](Z)(B)
\psset{linecolor=blue}%
% sqrt(0.3*0.3 + 0.3*0.3) = 0.42 and 45 - 26.57 = 18.43
\psline(Z)([nodesep=0.3]{B}Z)([nodesep=0.42,angle=18.43]Z)
      ([nodesep=0.3]{C}Z)
```

55 Coils and zigzags

pst-coil

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:

coilwidth=*dim*	**Default: 1cm**
coilheight=*num*	**Default: 1**
coilarm=*dim*	**Default: .5cm**
coilaspect=*angle*	**Default: 45**
coilinc=*angle*	**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

---

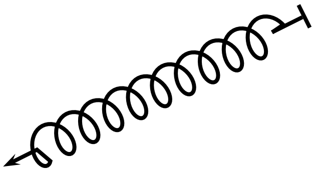
[40] For such Euclidian geometric drawings, the 'pst-eucl' contribution package from Dominique Rodriguez [59] offer many high level and powerful functions.

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.

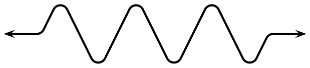
\pscoil and **\pszigzag** connect (*x0*,*y0*) and (*x1*,*y1*), starting and ending with straight line segments of length **coilarmA** and **coilarmB**, resp. Setting **coilarm** is the same as setting **coilarmA** and **coilarmB**.

Here is an example of **\pscoil**:



```
\pscoil[coilarm=.5cm,linewidth=1.5pt,coilwidth=.5cm]{<-|}(4,2)
```

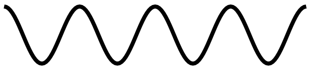
Here is an example of **\pszigzag**:



```
\pszigzag[coilarm=.5,linearc=.1]{<->}(4,0)
```

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:



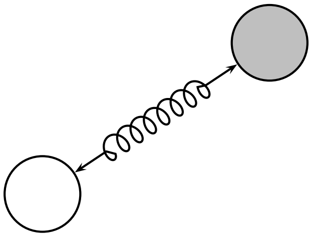
```
\psCoil[coilaspect=0,coilheight=1.33,
  coilwidth=.75,linewidth=1.5pt]{0}{1440}
```

pst-node

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:

\nccoil*[*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:



```
\cnode(.5,.5){.5}{A}
\cnode[fillstyle=solid,fillcolor=lightgray](3.5,2.5){.5}{B}
\nccoil[coilwidth=.3]{<->}{A}{B}
```

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 LATEX 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 commands[41]

\overlaybox
stuff
\endoverlaybox

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

\putoverlaybox{*string*}

then prints overlay *string*.

Here is an example:

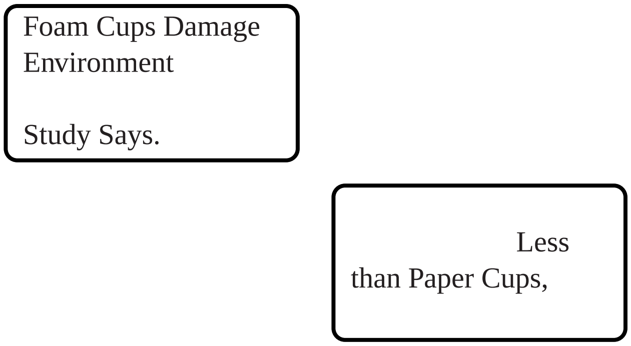
```
\overlaybox
  \psoverlay{all}
  \psframebox[framearc=.15,linewidth=1.5pt]{%
    \psoverlay{main}
```

---

[41] LATEX users can instead write:

`\begin{overlaybox}` *stuff* `\end{overlaybox}`

```
    \parbox{3.5cm}{\raggedright
      Foam  Cups  Damage  Environment  {\psoverlay{one}  Less
      than  Paper  Cups,}  Study  Says.}}
\endoverlaybox

\putoverlaybox{main}\hspace{.5in}\putoverlaybox{one}
```



Driver notes: Overlays use **\pstVerb** and **\pstverbscale**.

57 The gradient fill style

pst-grad

The file pst-grad.tex/pst-grad.sty, along with the PostScript header file pst-grad.pro, defines the gradient **fillstyle**, for gradiated 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 gradiation, 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*.

---

[42] The '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:

```
gradbegin  =  0  0.1  0.95
gradend    =  0  1      1
```

Here are two ways to change the gradient colors:

```
\psset{gradbegin=blue}
```

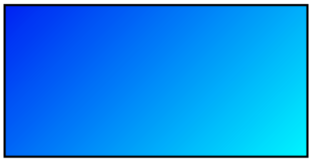
or, with PLAIN TeX and LaTeX (using the ‘color’ package):

```
\definecolor{rgb}{gradbegin}{1,0.4,0}
```

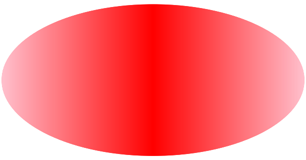
and with ConTeXt:

```
\definecolor{gradbegin}{r=1,g=0.4,b=0}
```

Try these examples:



```
\psframe[fillstyle=gradient,gradmidpoint=1,gradangle=45](4,2)
```



```
\psellipse[linestyle=none,fillstyle=gradient,gradmidpoint=0.5,
   gradangle=90,gradbegin=Pink,gradend=red](2,1)(2,1)
```

58 Typesetting text along a path

pst-text

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 **\pstheader** 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.

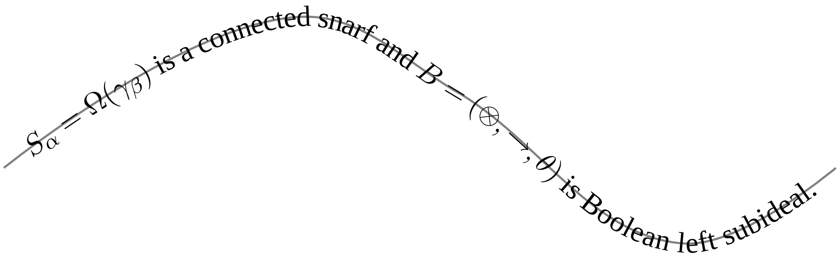
The default is l.

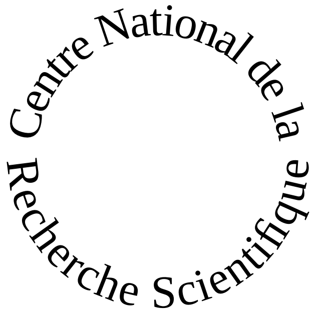
(<x>,<y>) is an offset. Characters are shifted by distance *x* along path, and are shifted up by *y*. "Up" means with respect to the path, at whatever point on the path corresponding to the middle of the character. (<x>,<y>) must be Cartesian coordinates. Both coordinates use **\psunit** as the default. The default coordinate is (0,\TPoffset), where **\TPoffset** a command whose default value is -.7ex. 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:

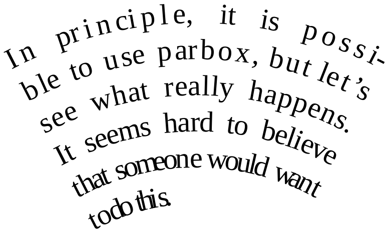
- Like with **\rput** and the graphics objects, it is up to you to leave space for **\pstextpath**.
- Results are unpredictable if *text* is wider than length of path.
- **\pstextpath** leaves the typesetting to TeX. It just intercepts the show operator to remap the coordinate system.

```
\pstextpath[c]%
  {\pscurve[linecolor=gray](0,1)(4,3)(6,2)(9,0)(11,1)}%
  {$S_\alpha=\Omega(\gamma_\beta)$ is a connected snarf and
   $B=(\otimes,\rightarrow,\theta)$ is Boolean left subideal.}
```


$S_\alpha = \Omega(\gamma_\beta)$ is a connected snarf and $B = (\otimes, \rightarrow, \theta)$ is Boolean left subideal.



```
\LARGE
\psset{linestyle=none}%
\pstextpath[c]{\psarcn(0,0){1.8}{180}{0}}{Centre National de la}
\pstextpath[c]{\psarc(0,0){1.8}{180}{0}}{Recherche Scientifique}
```



```
\pstextpath[c](0,0){\psarcn[linestyle=none](0,-6){4}{180}{0}}%
  {\parbox{4cm}{%
    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.}}
```

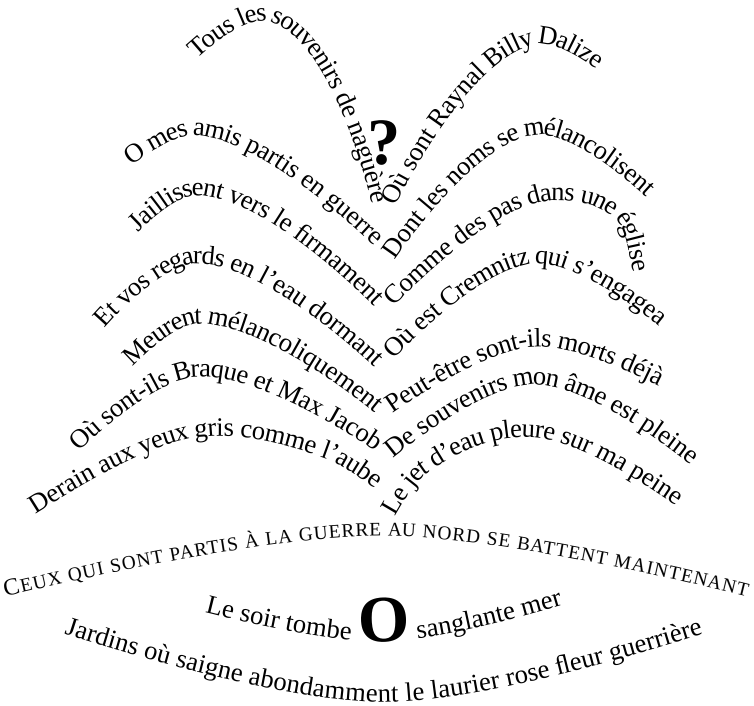
And, last, a peom of the *calligramme* form, *La colombe poignardée et le jet d'eau*, by the french poet Guillaume Appolinaire [3], re-typeset from the original[43] and first published in *Les Cahiers GUTenberg* in 1994 [22].

```
\small
\psset{linestyle=none}%
\pstextpath[r]{\pscurve(1.5,8)(3,9)(4.45,6.5)}
  {Tous les souvenirs de nagu^^e8re}
\pstextpath[r]{\pscurve(1,7)(2,7.5)(4.45,6)}
  {O mes amis partis en guerre}
\pstextpath[r]{\pscurve(1,6)(2,6.7)(4.45,5.2)}
  {Jaillissent vers le firmament}
% ...
```

---

[43] Only the code for few verses is shown here, but the complete code will be found in the source file of this documentation.

```
\pstextpath[c]{\pscurve(2.1,1.2)(4.5,0.8)(6.9,1.3)}
   {Le  soir  tombe  \textbf{\Huge  O}  sanglante  mer}
\pstextpath[c]{\pscurve(0,1)(4.5,0)(9,1)}
   {Jardins  o^^f9  saigne  abondamment  le  laurier  rose  fleur  guerri^^e8re}
```


Tous les souvenirs de naguère
?
Où sont Raynal Billy Dalize
O mes amis partis en guerre
Dont les noms se mélancolisent
Jaillissent vers le firmament
Comme des pas dans une église
Et vos regards en l'eau dormant
Où est Cremnitz qui s'engagea
Meurent mélancoliquement
Peut-être sont-ils morts déjà
Où sont-ils Braque et Max Jacob
De souvenirs mon âme est pleine
Derain aux yeux gris comme l'aube
Le jet d'eau pleure sur ma peine
CEUX QUI SONT PARTIS À LA GUERRE AU NORD SE BATTENT MAINTENANT
Le soir tombe O sanglante mer
Jardins où saigne abondamment le laurier rose fleur guerrière

59 Stroking and filling character paths

pst-char

The file pst-char.tex/pst-char.sty defines the command:

\pscharpath*[*par*]**{*text*}**

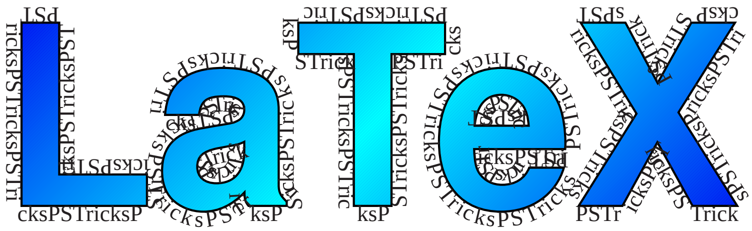
It strokes and fills the *text* character paths using the PSTricks **linestyle** and **fillstyle**.

The restrictions on DVI-to-PS drivers listed on page 152 for **\pstextpath** apply to **\pscharpath**. Furthermore, only outline (PostScript) fonts are affected.

```
\DeclareFixedFont{\Sf}{T1}{phv}{b}{n}{3.5cm}
\DeclareFixedFont{\Rm}{T1}{ptm}{m}{n}{3mm}

\pscharpath[linestyle=none,fillstyle=gradient,gradmidpoint=0.5,
              gradbegin=PaleGreen,gradend=ForestGreen]{\Sf  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**.

```
\pstextpath(0,-2mm){\pscharpath*[fillstyle=gradient,
                                gradangle=45,gradmidpoint=0.5]{\Sf  LaTeX}}%
                              {\Rm\multido{\i=1+1}{60}{PSTricks}}
```



The command[44]

\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]

```
\begin{pscharclip}[linewidth=3pt]{\rput[t](0,0){\Sf  ConTeXt}}
   \rput{58}{%
      \color{red}
      \begin{minipage}{11cm}
         \multido{\i=1+1}{220}{PSTricks  }
      \end{minipage}}
\end{pscharclip}
```

---

[44] LaTeX users can instead write:

`\begin{pscharclip}*[par]{text}  ...  \end{pscharclip}`

[45] Another way to do this is to use the **\pscharpath** macro with the **\psboxfill** one (see Section 50.3.)

ConTeXt

60 Importing EPS files

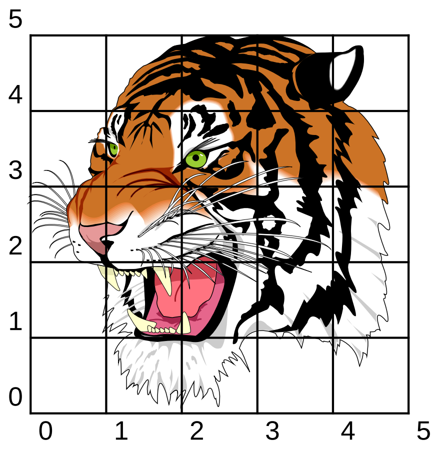
PSTricks does not come with any facility for including Encapsulated PostScript files, because there are other very good and well-tested macros for exactly that, specially the graphicx package for LaTeX, usable also with TeX.

What PSTricks *is* good for is embellishing your EPS picture. You can include an EPS file in the argument of **\rput**, as in

```
\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 TeX's point of view.[46]

```
\begin{pspicture}(5,5)
  \rput[bl](0,0){\includegraphics[scale=0.25]{tiger}}
  \psgrid[subgriddiv=0]
\end{pspicture}
```



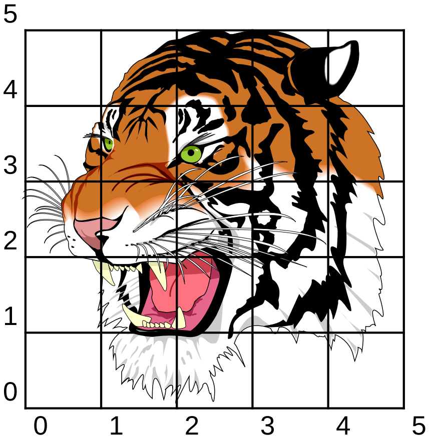
If you do not want to adjust the dimension of the pspicture environments by tries and errors, you can store the image in a TeX box (\pst@boxg is

---

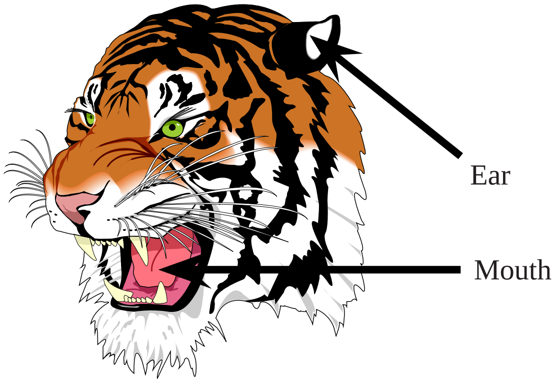
[46] Of course, with ConTeXt, use the PLAIN TeX syntax for the **\pspicture** environment and the macro \externalfigure.

the name of an internal temporary PSTricks box), and ask for it width and height:

```
\savebox{\pst@boxg}{\includegraphics[scale=0.25]{tiger}}
\begin{pspicture}(\wd\pst@boxg,\ht\pst@boxg)
   \rput[bl](0,0){\usebox{\pst@boxg}}
   \psgrid[subgriddiv=0]
\end{pspicture}
```



```
\begin{pspicture}(5,5)
   \rput[bl](0,0){\includegraphics[scale=0.25]{tiger}}
   \psset{linewidth=0.1,arrows=->,arrowscale=2}%
   \psline(6,3)(4,4.6)\uput[-45](6,3){Ear}
   \psline(6,1.5)(2,1.5)\uput[0](6,1.5){Mouth}
\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

```
\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

pst-eps

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 suports 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 problems[47]:

\TeXtoEPS
  stuff
\endTeXtoEPS

---

[47] LaTeX 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

```
%%BoundingBox llx 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=*dim*	**Default: 0pt**
bblly=*dim*	**Default: 0pt**
bburx=*dim*	**Default: 0pt**
bbury=*dim*	**Default: 0pt**

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=*none/new/all/all** **Default: new**

This parameter determines which **\PSTtoEPS** commands just skip over their arguments, and which create files, as follows:

---

[48] See 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.

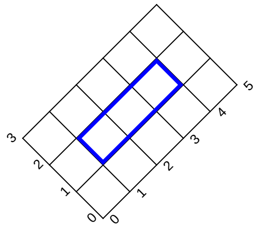
headerfile=*files* **Default:**

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:



```
\PSTtoEPS[bbllx=-0.2,bblly=-0.2,bburx=5,bbury=3]{Frame.eps}{%
  \psgrid[subgriddiv=0](5,3)
  \psframe[linecolor=blue,linewidth=0.1](1,1)(4,2)}

\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.)

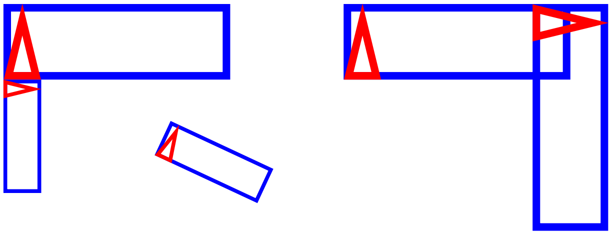
Rotation=*num* **Default: none**

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.)

```
\psNewHeader{Frame.pro}{%
  \psDefineProc{Frame}{%
    \psset{linewidth=0.1}%
    \psframe[linecolor=blue](3,1)
    \pstriangle[linecolor=red](0.25,0)(0.5,1)}}

\psUseHeader{Frame.pro}

% ...

%\begin{document}

% ...

\psUseProc{Frame}
\psUseProc[Rotation=-90,Scale=0.5]{Frame}
\psUseProc[Translation={4.5,0}]{Frame}
\psUseProc[Translation={2,-1},Rotation=-25,Scale=0.5]{Frame}
\psUseProc[Translation={7,1},Rotation=-90]{Frame}
```



XII Writing high-level macros[49]

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

pst-key

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 TeX, LaTeX and ConTeXt. Developers are strongly encouraged to create packages that are *generic*, not limited to LaTeX or another TeX 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

```
\setkeys{psset}{param1=value1,...,paramN=valueN}
```

will initialize the defined parameters, inside the TeX 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):

```
\setkeys{psset}{...}% Assignment of global parameters

\def\PstXxxx{\def\pst@par{}\pst@object{PstXxxx}}

\def\PstXxxx@i{{% Two braces, because parameter
```

---

[49] This chapter was written in 2003 by Denis Girou.

```
                     % assignments must be local
\use@par% Assignment of local parameters
% The code itself, which can of course include calls
% to auxiliary macros
% ...
}}
```

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:

```
\setkeys{psset}{...}% Assignment of global parameters

\def\PstXxxx{%
% Test for optional parameters
\@ifnextchar[{\PstXxxx@i}{\PstXxxx@i[]}}

\def\PstXxxx@i[#1]{{% Two braces, because parameter
                          % assignments must be local
\setkeys{psset}{#1}% Assignment of local parameters
% ...
}}
```

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 practive 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. TEX 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@cnta,…,\pst@cnth) and six temporary dimension registers (\pst@dima,…,\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 scare resource. See the examples below and the PSTricks code itself for more details.

```
\define@key{psset}{Integer}{\pst@getint{#1}{\PstXxxx@Integer}}
```

Here is an example which prints a partial multiplication table.

```
% “Start”, “End” and “Value” parameters
\define@key{psset}{Start}{\pst@getint{#1}{\MultTable@Start}}
\define@key{psset}{End}{\pst@getint{#1}{\MultTable@End}}
\define@key{psset}{Value}{\pst@getint{#1}{\MultTable@Value}}
\setkeys{psset}{Start=1,End=5,Value=2}% Default values

% Main macro for “multiplication table” object
\def\MultTable{\def\pst@par{}\pst@object{MultTable}}

\def\MultTable@i{{%
\use@par% Assignment of local parameters
\pst@cnth=\MultTable@End
\advance\pst@cnth-\MultTable@Start
\advance\pst@cnth\@ne
\multido{\iValue=\MultTable@Start+\@ne}{\pst@cnth}{%
   \pst@cntg=\iValue
   \multiply\pst@cntg\MultTable@Value
   \iValue$\times$\MultTable@Value = \the\pst@cntg
   \ifnum\multidocount=\pst@cnth\else; \fi}}}

\MultTable

\MultTable[Value=9]

\setkeys{psset}{Value=6}

\MultTable[Start=6,End=11]

\MultTable[Start=19742,End=19742]
```

1×2 = 2; 2×2 = 4; 3×2 = 6; 4×2 = 8; 5×2 = 10

1×9 = 9; 2×9 = 18; 3×9 = 27; 4×9 = 36; 5×9 = 45

6×6 = 36; 7×6 = 42; 8×6 = 48; 9×6 = 54; 10×6 = 60; 11×6 = 66

19742×6 = 118452

We can also handle a special behavior with the * convention. Here, the result of the multiplication is put in a colored circle.

```
\def\MultTable@i{{%
\use@par% Assignment of local parameters
\pst@cnth=\MultTable@End
\advance\pst@cnth-\MultTable@Start
\advance\pst@cnth\@ne
\multido{\iValue=\MultTable@Start+\@ne}{\pst@cnth}{%
   \pst@cntg=\iValue
   \multiply\pst@cntg\MultTable@Value
   \iValue$\times$\MultTable@Value  =
   \if@star
      {\pscirclebox*[fillstyle=solid,fillcolor=red,framesep=0.02]{%
         \textcolor{yellow}{\the\pst@cntg}}}%
   \else
      \the\pst@cntg
   \fi
   \ifnum\multidocount=\pst@cnth\else;  \fi}}}

\MultTable[Value=13,Start=7,End=10]

\MultTable*[Value=13,Start=7,End=10]
```

7×13 = 91; 8×13 = 104; 9×13 = 117; 10×13 = 130

7×13 = 91; 8×13 = 104; 9×13 = 117; 10×13 = 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:

```
\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 TEX 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.

```
\define@key{psset}{Length}{%
\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

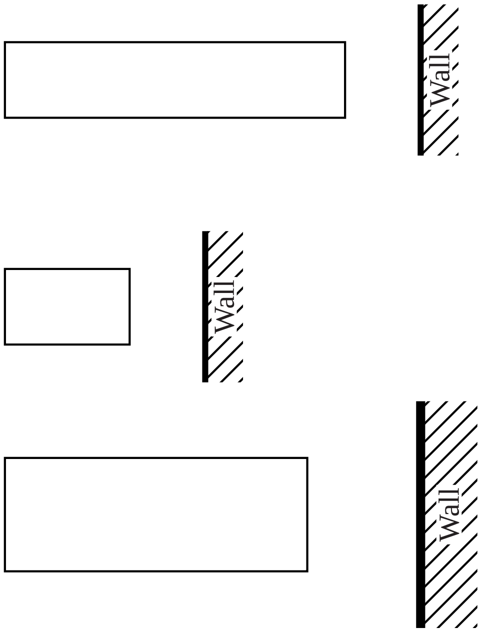
```
\setkeys{psset}{Length=3}
...
\PstXxxx[unit=0.2]
```

will use a length of 1cm, as it assignemt as been done before. So, if such order of assignement is expected, it it 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.

```
\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 explicitely 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.



```
% “Length” parameter: length of the bar
\define@key{psset}{Length}{\edef\PstBar@Length{#1}}
\setkeys{psset}{Length=3}% Default value

% Main macro for “bar” object
\def\PstBar{\def\pst@par{}\pst@object{PstBar}}

\def\PstBar@i{{%
\setkeys{psset}{framesep=1pt}%
\use@par% Assignment of local parameters
\psframe(\PstBar@Length,\@ne)
\pssetlength{\pst@dima}{\PstBar@Length}%
\psaddtolength{\pst@dima}{\@ne}%
\psline[linewidth=0.08](\pst@dima,-0.5)(\pst@dima,1.5)
\pst@dimb=\pst@dima
\psaddtolength{\pst@dimb}{0.5}%
\psframe[linestyle=none,fillstyle=hlines]
            (\pst@dima,-0.5)(\pst@dimb,1.5)
\psaddtolength{\pst@dima}{0.25}%
\rput*{90}(\pst@dima,0.5){Wall}}}

\rput(-2,3){\PstBar[Length=4.5]}
\rput(-2,0){\PstBar[Length=1.65cm]}
\rput(-2,-3){\PstBar[unit=1.5,Length=4cm]}
```

63.3 String parameters

String parameters should also be stored as personal macros using either:

```
\define@key{psset}{String}{\edef\PstXxxx@String{#1}}
```

or

```
\define@key{psset}{String}{\def\PstXxxx@String{#1}}
```

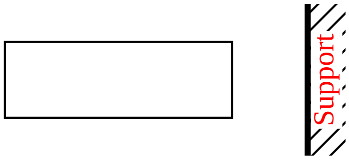
If the value of `#1` can be evaluated correctly at the time of definition, the `\edef` alternative should be used, because it avoid side effects due to later expansion, as in the following case where the `\PstObject` called in the last line will have `OLD` for value of the `String` parameter if `\edef` 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.

```
\def\MyString{OLD}
\setkeys{psset}{String=\MyString}
...
\def\MyString{NEW}
\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):

```
\define@key{psset}{String}{%
\def\PstXxxx@Macro{\@nameuse{PstXxxx@Macro@#1}}}

\setkeys{psset}{String=BBBB}% Default value

% Macros for possible values
\def\PstXxxx@Macro@AAAA{...}
\def\PstXxxx@Macro@BBBB{...}
\def\PstXxxx@Macro@CCCC{...}

\def\PstXxxx{\def\pst@par{}\pst@object{PstXxxx}}

\def\PstXxxx@i{{%
% ...
\expandafter\PstXxxx@Macro
% ...
}}
```



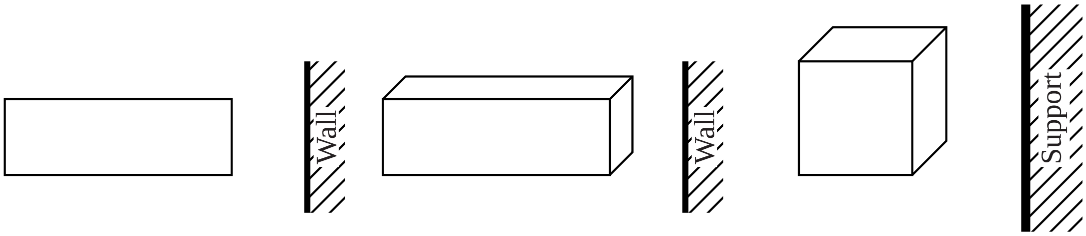
```
% “Label” parameter: label to legend the support
\define@key{psset}{Label}{\def\PstBar@Label{#1}}
\setkeys{psset}{Label=Wall}% Default value

\def\PstBar@i{{%
% ...
\rput*{90}(\pst@dima,0.5){\PstBar@Label}}}

\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.

```
\newif\ifPstXxxx@Boolean
\define@key{psset}{Boolean}[true]{%
\@nameuse{PstXxxx@Boolean#1}}
```

```
% “ThreeD” parameter: to emulate or not a 3d bar
\newif\ifPstBar@ThreeD
\define@key{psset}{ThreeD}[true]{%
\@nameuse{PstBar@ThreeD#1}}

\setkeys{psset}{ThreeD=false}% Default value

\def\PstBar@i{{%
\setkeys{psset}{framesep=1pt}%
\use@par% Assignment of local parameters
\psframe(\PstBar@Length,\@ne)
\pssetlength{\pst@dima}{\PstBar@Length}%
\ifPstBar@ThreeD
   \pst@dimb=\pst@dima
   \psaddtolength{\pst@dimb}{0.3}%
   \psline(0,1)(0.3,1.3)(\pst@dimb,1.3)
            (\pst@dimb,0.3)(\PstBar@Length,0)
   \psline(\PstBar@Length,1)(\pst@dimb,1.3)
\fi
\psaddtolength{\pst@dima}{\@ne}%
\psline[linewidth=0.08](\pst@dima,-0.5)(\pst@dima,1.5)
\pst@dimb=\pst@dima
\psaddtolength{\pst@dimb}{0.5}%
\psframe[linestyle=none,fillstyle=hlines]
            (\pst@dima,-0.5)(\pst@dimb,1.5)
\psaddtolength{\pst@dima}{0.25}%
\rput*{90}(\pst@dima,0.5){\PstBar@Label}}}

\rput(-7.5,0){\PstBar}
\rput(-2.5,0){\PstBar[ThreeD=true]}
\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 (*x0*,*y0*)(*x1*,*y1*)… (*xn*,*yn*) 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:

```
\begin@OpenObj  ...  \end@OpenObj
\begin@AltOpenObj  ...  \end@OpenObj
\begin@ClosedObj  ...  \end@ClosedObj
\begin@SpecialObj  ...  \end@SpecialObj
```

We will give examples for the first, third and fourth ones (the second is obviousy very similar to the first one.)

First we define a real value parameter which is used to control the extent of the randomness effect:

```
%  Random  coefficient  parameter
\define@key{psset}{RandomCoefficient}{%
\pst@checknum{#1}{\PstRandomLine@RandomCoefficient}}

\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

---

[50] For complementary information, see in [79] the detailed explanation of how the **\pspolygon** macro is defined.

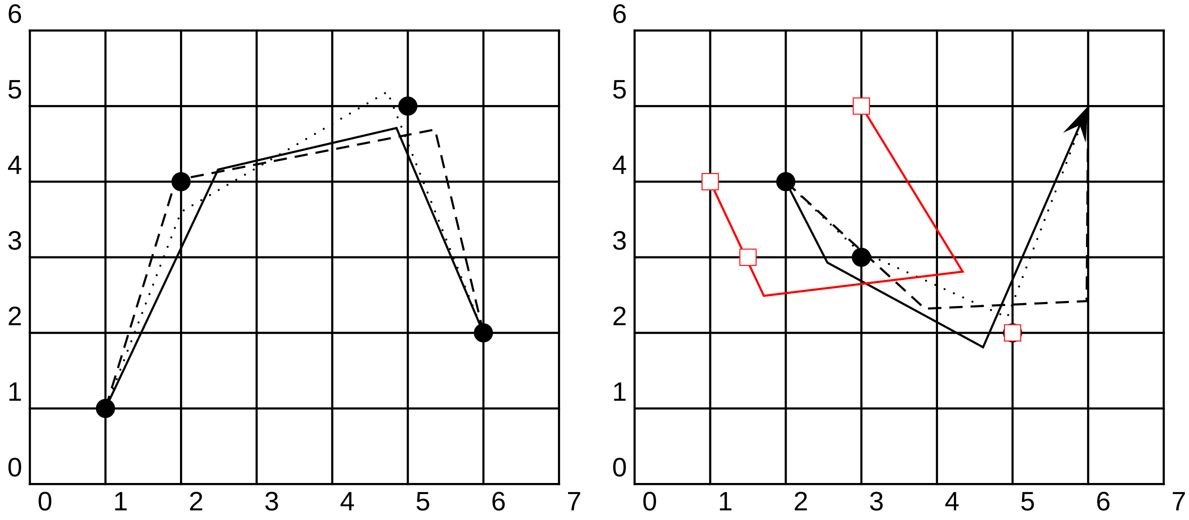
\pst@getarrows macro. Then the \begin@OpenObj macro initializes various things for this kind of object and the \pst@getcoors macro allows management of a sequence of coordinates of unknown length, which could simply be cartesian but also can be any kind of the *special* coordinates, accumulating their 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 **linearc** 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.

```
\def\PstRandomLine{\pst@object{PstRandomLine}}

\def\PstRandomLine@i{%
\pst@getarrows{%
\begin@OpenObj
\pst@getcoors[\PstRandomLine@ii}}

\def\PstRandomLine@ii{%
\addto@pscode{\pst@cp \PstRandomLine@iii \tx@Line}%
\end@OpenObj}

\def\PstRandomLine@iii{%
/RandomValue {rand 100 mod 100 div 0.5 sub
              \pst@number{\psunit} mul
              \PstRandomLine@RandomCoefficient\space mul} def
/Lineto {exch RandomValue add exch RandomValue add lineto}def
\ifshowpoints true \else false \fi}

\pspicture(7,6)\psgrid[subgriddiv=0]
```

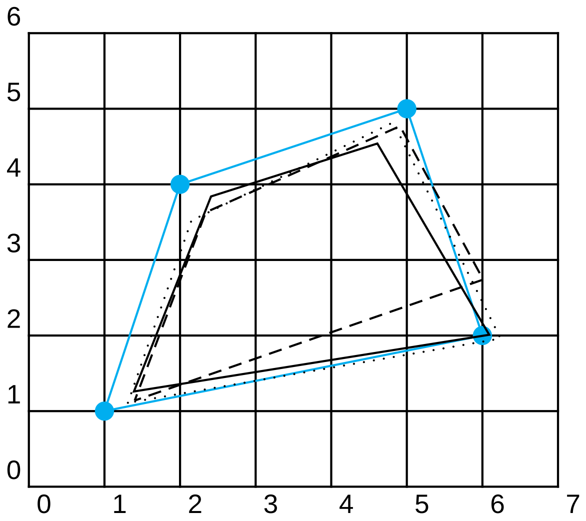
```
  \PstRandomLine[showpoints=true,dotscale=2](1,1)(2,4)(5,5)(6,2)
  \PstRandomLine[linestyle=dashed](1,1)(2,4)(5,5)(6,2)
  \PstRandomLine[linestyle=dotted](1,1)(2,4)(5,5)(6,2)
\endpspicture
\hfill
\pspicture(7,6)\psgrid[subgriddiv=0]
  \SpecialCoor
  \pnode(5,2){Node}
  \PstRandomLine[arrowscale=3,showpoints=true,dotscale=2]
                {->}(2,4)(3,3)(Node)(6,5)
  \PstRandomLine[linestyle=dashed,RandomCoefficient=2]
                (2,4)(3,3)(Node)(6,5)
  \PstRandomLine[linestyle=dotted,RandomCoefficient=0.5]
                (2,4)(3,3)(Node)(6,5)
  \PstRandomLine[xunit=0.5,linecolor=red,showpoints=true,
                 dotstyle=square,dotscale=2,
                 RandomCoefficient=3](2,4)(3,3)(Node)(6,5)
\endpspicture
```

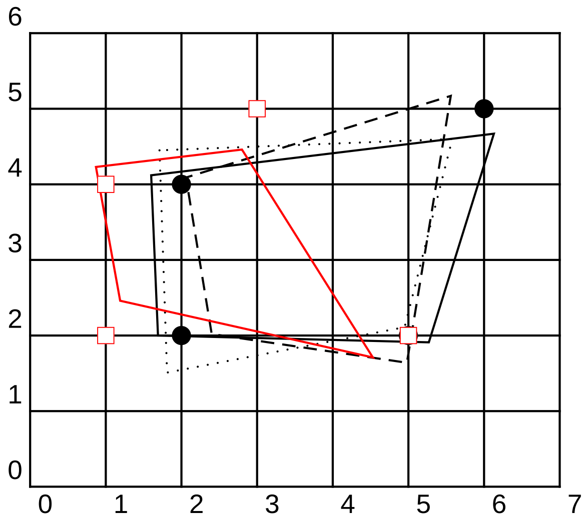


Closed objects will be illustrated by a variation of the \pspolygon nmacro. After dealing with the optional parameters, the \PstRandomPolygon@i macro initiates the closed object by calling the \begin@ClosedObj macro, then reading and accumulating the coordinates by the \pst@getcoors macro as in the preceding randomized line example. Then the \PstRandomPolygon@ii macro calls the Polygon PostScript operator (using the \tx@Polygon TeX 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.

```
\def\PstRandomPolygon{\pst@object{PstRandomPolygon}}

\def\PstRandomPolygon@i{%
\begin@ClosedObj
```

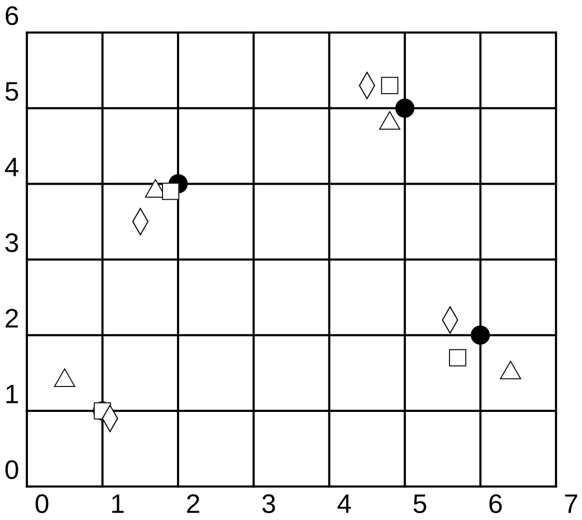
```
\def\pst@cp{}%
\pst@getcoors[\PstRandomPolygon@ii}

\def\PstRandomPolygon@ii{%
\addto@pscode{\PstRandomLine@iii  \tx@Polygon}%
\def\pst@linetype{1}%
\end@ClosedObj}

\pspicture(7,6)\psgrid[subgriddiv=0]
  \pspolygon[linecolor=cyan,showpoints=true,dotscale=2]
            (1,1)(2,4)(5,5)(6,2)
  \PstRandomPolygon(1,1)(2,4)(5,5)(6,2)
  \PstRandomPolygon[linestyle=dashed](1,1)(2,4)(5,5)(6,2)
  \PstRandomPolygon[linestyle=dotted](1,1)(2,4)(5,5)(6,2)
\endpspicture
\hfill
\pspicture(7,6)\psgrid[subgriddiv=0]
  \SpecialCoor
  \pnode(5,2){Node}
  \PstRandomPolygon[showpoints=true,dotscale=2](2,4)(2,2)(Node)
                    (6,5)
  \PstRandomPolygon[linestyle=dashed,RandomCoefficient=2]
                    (2,4)(2,2)(Node)(6,5)
  \PstRandomPolygon[linestyle=dotted,RandomCoefficient=0.5]
                    (2,4)(2,2)(Node)(6,5)
  \PstRandomPolygon[xunit=0.5,linecolor=red,showpoints=true,
                     dotstyle=square,dotscale=2,
                     RandomCoefficient=3](2,4)(2,2)(Node)(6,5)
\endpspicture
```

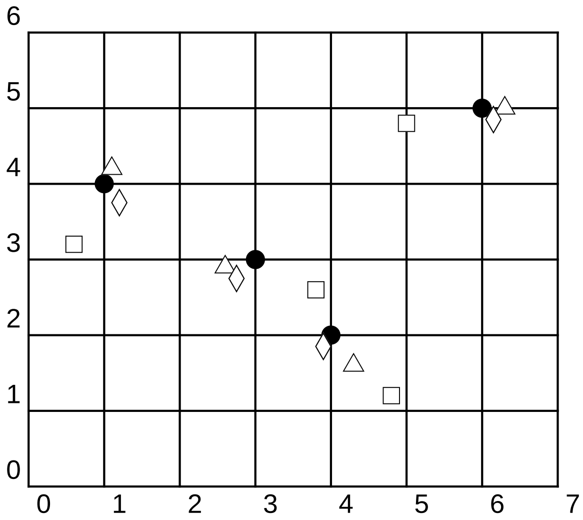




Special objects will be illustrated by a variation of the \psdots macro. After dealing with the optional parameters, the \PstRandomDots@i macro initiates the special object by calling the \begin@SpecialObj macro, then reading and accumulating the coordinates by the \pst@getcoors macro, as in the randomized 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.

```
\def\PstRandomDots{\pst@object{PstRandomDots}}

\def\PstRandomDots@i{%
\begin@SpecialObj
\pst@getcoors[\PstRandomDots@ii}

\def\PstRandomDots@ii{%
\addto@pscode{false \tx@NArray \PstRandomDots@iii}%
\end@SpecialObj}

\def\PstRandomDots@iii{%
\psk@dotsize
\@nameuse{psds@\psk@dotstyle}
newpath
/RandomValue {rand 10 mod 10 div 0.5 sub
              \pst@number{\psunit} mul
              \PstRandomLine@RandomCoefficient\space mul} def
n { transform floor 0.5 add exch floor 0.5 add exch itransform
    exch RandomValue add exch RandomValue add Dot } repeat}

\pspicture(7,6)\psgrid[subgriddiv=0]
  \psset{dotscale=2}%
  \psdots(1,1)(2,4)(5,5)(6,2)
  \PstRandomDots[dotstyle=triangle](1,1)(2,4)(5,5)(6,2)
  \PstRandomDots[dotstyle=square](1,1)(2,4)(5,5)(6,2)
  \PstRandomDots[dotstyle=diamond](1,1)(2,4)(5,5)(6,2)
\endpspicture
\hfill
\pspicture(7,6)\psgrid[subgriddiv=0]
  \SpecialCoor
  \psset{dotscale=2}%
  \pnode(4,2){Node}
  \psdots(1,4)(3,3)(Node)(6,5)
  \PstRandomDots[dotstyle=triangle](1,4)(3,3)(Node)(6,5)
  \PstRandomDots[dotstyle=square,RandomCoefficient=2]
                 (1,4)(3,3)(Node)(6,5)
  \PstRandomDots[dotstyle=diamond,RandomCoefficient=0.5]
                 (1,4)(3,3)(Node)(6,5)
\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:

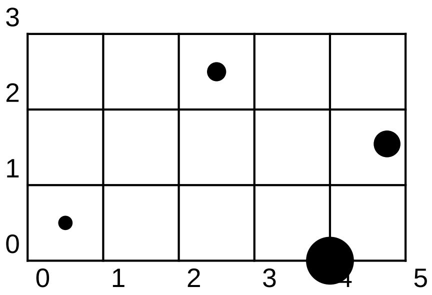
```
\PstDotsScaled(x0,y0,1.5)(x1,y1,2)(x2,y2,5)(x3,y3,2.8)
```

which could seem easier to read, must be avoided.

The \@ifnextchar macro allows determing 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.

```
\def\PstDotsScaled{\def\pst@par{}\pst@object{PstDotsScaled}}

\def\PstDotsScaled@i{%
\begingroup
\use@par% Assignment of local parameters
\PstDotsScaled@GetCoordinate}
```

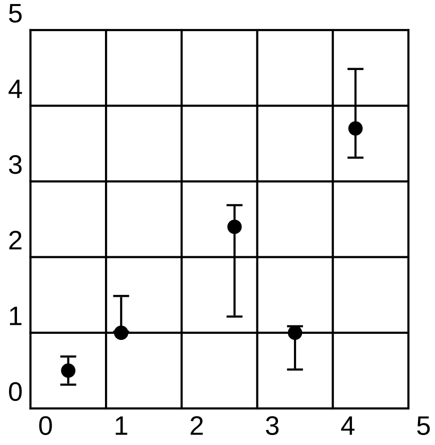
```

\def\PstDotsScaled@GetCoordinate(#1)#2{%
\psdot[dotscale=#2](#1)
\@ifnextchar({\PstDotsScaled@GetCoordinate}{\endgroup}}

\pspicture(5,3)\psgrid[subgriddiv=0]
   \SpecialCoor
   \pnode(2.5,2.5){Node}
   % “rand 5 mod” return an integer between 0 and 5 and
   % “rand 3 mod” return an integer between 0 and 3
   \PstDotsScaled(0.5,0.5){1.5}(Node){2}
                    (! rand 5 mod rand 3 mod){5}(5;18){2.8}
\endpspicture
```

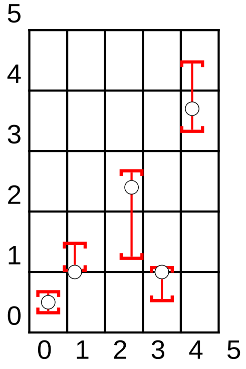


In the same way, we will now define a macro to draw a series of dots and an error bar for each of them, using two additional parameters.

```
% Style of error bars parameter
\define@key{psset}{StyleBars}{%
\edef\PstDataWithErrorBars@StyleBars{#1}}

\newpsstyle{StyleBars@Default}{arrows=|-|}
\setkeys{psset}{StyleBars=StyleBars@Default}% Default value

\def\PstDataWithErrorBars{%
\def\pst@par{}\pst@object{PstDataWithErrorBars}}

\def\PstDataWithErrorBars@i{%
\begingroup
\use@par% Assignment of local parameters
\PstDataWithErrorBars@GetCoordinate}

\def\PstDataWithErrorBars@GetCoordinate(#1,#2)#3#4{%
\PstDataWithErrorBars@DoCoordinate(#1,#2){#3}{#4}%
\@ifnextchar({\PstDataWithErrorBars@GetCoordinate}{\endgroup}}

\def\PstDataWithErrorBars@DoCoordinate(#1,#2)#3#4{%
\psline[style=\PstDataWithErrorBars@StyleBars]
       (! #1 #2 #3 add)(! #1 #2 #4 add)
\psdot(#1,#2)}

\SpecialCoor

```

```
\psset{dotscale=1.5}%

\pspicture(5,5)\psgrid[subgriddiv=0]
  \PstDataWithErrorBars(0.5,0.5){-0.2}{0.2}(1.2,1){0}{0.5}
    (2.7,2.4){-1.2}{0.3}(3.5,1){-0.5}{0.1}(4.3,3.7){-0.4}{0.8}
\endpspicture
\hfill
%
\newpsstyle{MyStyleBars}{linecolor=red,arrows=[-],arrowscale=1.5}
\psset{xunit=0.5,yunit=0.8}%
\pspicture(5,5)\psgrid[subgriddiv=0]
  \PstDataWithErrorBars[dotstyle=o,StyleBars=MyStyleBars]
    (0.5,0.5){-0.2}{0.2}(1.2,1){0}{0.5}
    (2.7,2.4){-1.2}{0.3}(3.5,1){-0.5}{0.1}(4.3,3.7){-0.4}{0.8}
\endpspicture
```

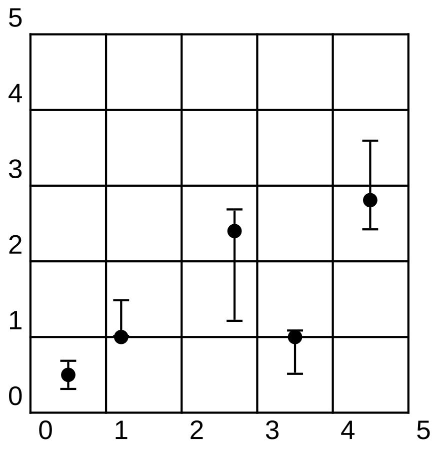


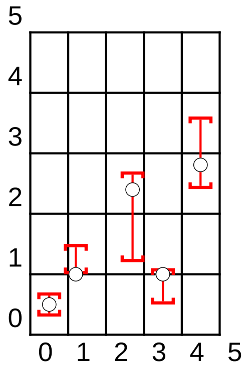


Here, we deal with each coordinate as two parameters separated by a comma, so this will work only for cartesian coordinates. To manage *special* ones, we must of course use only one argument, then call the \pst@getcoor macro which will return the two coordinates of the point, even if it is a node or 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.

```
\def\PstDataWithErrorBars@GetCoordinate(#1)#2#3{%
\PstDataWithErrorBars@DoCoordinate(#1){#2}{#3}%
\@ifnextchar({\PstDataWithErrorBars@GetCoordinate}{\endgroup}}

\def\PstDataWithErrorBars@DoCoordinate(#1)#2#3{%
\pst@getcoor{#1}{\pst@temph}%
\pst@Verb{%
  /XCoor {\pst@temph pop \pst@number{\psxunit} div} def
  /YCoor {\pst@temph exch pop \pst@number{\psyunit} div} def}
\psline[style=\PstDataWithErrorBars@StyleBars]
```

```
        (! XCoor YCoor #2 add)(! XCoor YCoor #3 add)
\psdot(#1)}

\SpecialCoor

\psset{dotscale=1.5}%

\pspicture(5,5)\psgrid[subgriddiv=0]
    \pnode(2.7,2.4){Node}
    \PstDataWithErrorBars(0.5,0.5){-0.2}{0.2}(1.2,1){0}{0.5}
        (Node){-1.2}{0.3}(! 2 1.5 add 1){-0.5}{0.1}
        (5.3;32){-0.4}{0.8}
\endpspicture
\hfill
%
\newpsstyle{MyStyleBars}{linecolor=red,arrows=[-],arrowscale=1.5}
\psset{xunit=0.5,yunit=0.8}%
\pspicture(5,5)\psgrid[subgriddiv=0]
    \pnode(2.7,2.4){Node}
    \PstDataWithErrorBars[dotstyle=o,StyleBars=MyStyleBars]
        (0.5,0.5){-0.2}{0.2}(1.2,1){0}{0.5}(Node){-1.2}{0.3}
        (! 2 1.5 add 1){-0.5}{0.1}
        (! 5.3 32 cos mul 5.3 32 sin mul){-0.4}{0.8}
\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):

```
0.7 4.1 -0.7 0.6
2.6 3.4 -2.1 0.2
3.2 2   -1.2 0
4.1 4.6 -0.3 0.2
4.6 1.4 -0.5 1.8
```

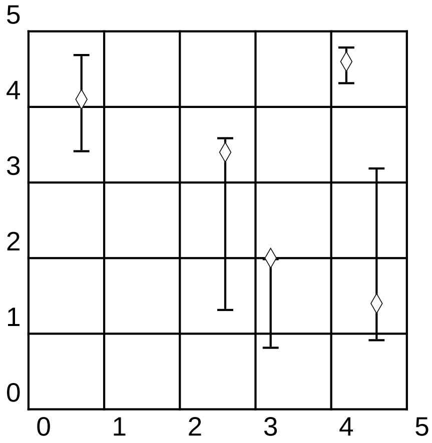
or

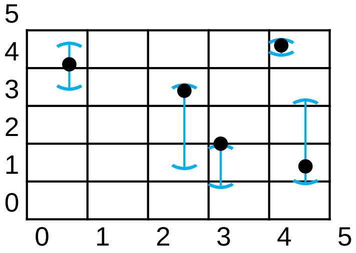
```
(0.7,  4.1,  -0.7,  0.6)
(2.6,  3.4,  -2.1,  0.2)
(3.2,  2,    -1.2,  0)
(4.1,  4.6,  -0.3,  0.2)
(4.6,  1.4,  -0.5,  1.8)
```

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.

```
\def\PstDataFileWithErrorBars{%
\def\pst@par{}\pst@object{PstDataFileWithErrorBars}}

\def\PstDataFileWithErrorBars@i#1{%
\begingroup
\use@par% Assignment of local parameters
% Read the data from the file and put them in a macro
\readdata{\PstDataFileWithErrorBars@Data}{#1}%
\PstDataFileWithErrorBars@GetCoordinate{%
   \PstDataFileWithErrorBars@Data}}

\def\PstDataFileWithErrorBars@GetCoordinate#1{%
\expandafter\PstDataFileWithErrorBars@GetCoordinate@i#1  }

\def\PstDataFileWithErrorBars@GetCoordinate@i#1{%
\PstDataFileWithErrorBars@GetCoordinate@ii#1}

\def\PstDataFileWithErrorBars@GetCoordinate@ii%
   #1 #2 #3 #4 #5 #6 #7 #8 {%
\PstDataFileWithErrorBars@DoCoordinate(#2,#4){#6}{#8}%
\@ifnextchar D{\PstDataFileWithErrorBars@GetCoordinate@ii}%
                  {\endgroup}}

\let\PstDataFileWithErrorBars@DoCoordinate
   \PstDataWithErrorBars@DoCoordinate

\psset{dotscale=1.5}%

\pspicture(5,5)\psgrid[subgriddiv=0]
   \PstDataFileWithErrorBars[dotstyle=diamond]{err-bars.dat}
\endpspicture
\hfill
%
\newpsstyle{MyStyleBars}{linecolor=cyan,arrows=(-),arrowscale=1.5}
```

```
\psset{xunit=0.8,yunit=0.5}
\pspicture(5,5)\psgrid[subgriddiv=0]
  \PstDataFileWithErrorBars[StyleBars=MyStyleBars]{err-bars.dat}
\endpspicture
```





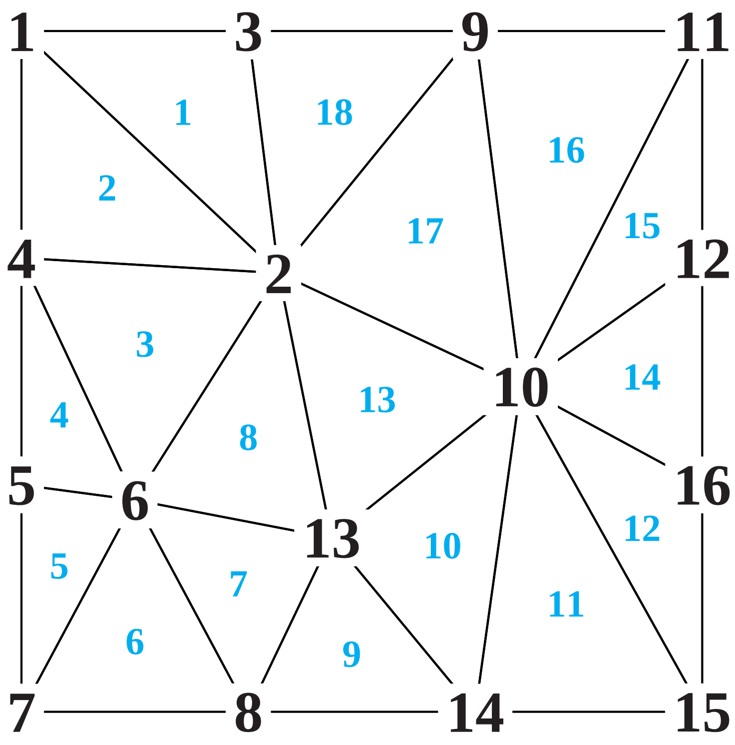
We finish this section with a more sophisticated example, which draws a general mesh based on triangles, as used in finite elements numerical methods, with an automatic computation of the barycenters of the triangles to position their numbers. We want of course to be as generic as possible, just defining 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.

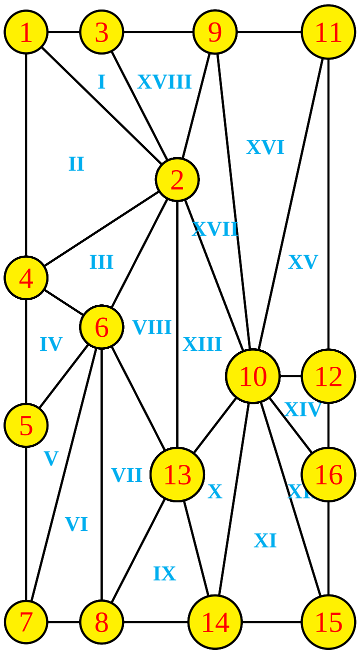
Note that we store the definition of the points as macros, whose names are suffixed by the rank of the point in the list of coordinates. We do not need here to store them as PSTricks nodes, which would make the resulting PostScript code larger and slower without any benefit.

```
\def\PstMeshNodes{%
\pst@cnth=\z@
\PstMeshNodes@GetCoordinate}

\def\PstMeshNodes@GetCoordinate(#1){%
\advance\pst@cnth\@ne
\@namedef{MeshNode\the\pst@cnth}{#1}%
\@ifnextchar({\PstMeshNodes@GetCoordinate}{}}

\def\PstMeshTriangles{\def\pst@par{}\pst@object{PstMeshTriangles}}

\def\PstMeshTriangles@i{%
\begingroup
\use@par% Assignment of local parameters
% PostScript macro to help to compute the barycenter
% of the three points
\pst@Verb{/ForBarycenter {add 2 div exch dup /X3 ED sub
                          4 1 roll
                          add 2 div neg exch dup /X6 ED
```

```
                       add div dup X6 mul X3 add} def}%
\PstMeshTriangles@GetTriangle}

\def\PstMeshTriangles@GetTriangle(#1,#2,#3)#4{%
% #1, #2, #3 : first, second and third point numbers
% #4 : label to print on the barycenter of this triangle
\pspolygon(\@nameuse{MeshNode#1})(\@nameuse{MeshNode#2})
         (\@nameuse{MeshNode#3})
\PstMeshPrintNode{\@nameuse{MeshNode#1}}{#1}%
\PstMeshPrintNode{\@nameuse{MeshNode#2}}{#2}%
\PstMeshPrintNode{\@nameuse{MeshNode#3}}{#3}%
\PstMeshTrianglesLabelOnBarycenter{\@nameuse{MeshNode#1}}
  {\@nameuse{MeshNode#2}}{\@nameuse{MeshNode#3}}{#4}%
\@ifnextchar({\PstMeshTriangles@GetTriangle}{\endgroup}}

\def\PstMeshTrianglesLabelOnBarycenter#1#2#3#4{%
% #1 : coordinates of the first point (A) of the triangle
% #2 : coordinates of the second point (B) of the triangle
% #3 : coordinates of the third point (C) of the triangle
% #4 : label to print on the barycenter
\pst@getcoor{#1}{\pst@tempa}%
\pst@getcoor{#2}{\pst@tempb}%
\pst@getcoor{#3}{\pst@tempc}%
\rput(! % To retrieve the X and Y coordinates of the three points
      \pst@tempa \pst@number{\psyunit} div /YA ED
                 \pst@number{\psxunit} div /XA ED
      \pst@tempb \pst@number{\psyunit} div /YB ED
                 \pst@number{\psxunit} div /XB ED
      \pst@tempc \pst@number{\psyunit} div /YC ED
                 \pst@number{\psxunit} div /XC ED
      XA XB add 2 div XC eq
         {XA XC XB YA YC YB ForBarycenter}
         {XC XB XA YC YB YA ForBarycenter} ifelse
      /ZB ED /ZA ED
      XA XC add 2 div XB eq
         {XA XC XB YA YC YB ForBarycenter}
         {XB XC XA YB YC YA ForBarycenter} ifelse
      /ZD ED /ZC ED
      % X = (ZB - ZD) / (ZA - ZC) and Y = ZD - ZC * X
      ZB ZD sub ZA ZC sub div dup ZC mul neg ZD add)
    {\PstMeshPrintLabel{#4}}}

% Default format to print the mesh nodes
\def\PstMeshPrintNode#1#2{\rput*(#1){\bf\scalebox{2}{#2}}}

% Default format to print the mesh labels on the barycenters
\def\PstMeshPrintLabel#1{\it\scalebox{2}{#1}}

% Usage examples
% ============

% Definition of the nodes of the mesh
\PstMeshNodes(0,9)(3.4,5.8)(3,9)(0,6)(0,3)(1.5,2.8)(0,0)(3,0)
```

```
                (6,9)(6.6,4.3)(9,9)(9,6)(4.1,2.3)(6,0)(9,0)(9,3)

% To draw the mesh
\newcommand{\PstMeshTrianglesDraw}[1][]{%
\PstMeshTriangles[#1](1,2,3){1}(1,2,4){2}(2,4,6){3}(4,5,6){4}(5,6,7){5}
   (6,7,8){6}(6,8,13){7}(2,6,13){8}(8,13,14){9}(10,13,14){10}
   (10,14,15){11}(10,15,16){12}(2,10,13){13}(10,12,16){14}
   (10,11,12){15}(9,10,11){16}(2,9,10){17}(2,3,9){18}}

% New format for the label
\renewcommand{\PstMeshPrintLabel}[1]{%
   \Large\textbf{\textcolor{cyan}{#1}}}

\PstMeshTrianglesDraw
```





```
% New definition of the nodes of the mesh
\PstMeshNodes(0,12)(4,9)(2,12)(0,7)(0,4)(2,6)(0,0)(2,0)
                         (5,12)(6,5)(8,12)(8,5)(4,3)(5,0)(8,0)(8,3)

\renewcommand{\PstMeshPrintNode}[2]{%
\rput(#1){%
   \pscirclebox[fillstyle=solid,fillcolor=yellow]{\textcolor{red}{#2}}}}

\makeatletter
\renewcommand{\PstMeshPrintLabel}[1]{%
\scriptsize\textbf{\textcolor{cyan}{\@Roman{#1}}}}
\makeatother

% We use the mesh previously defined
\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]

```
\def\FileVersion{1.2}
\def\FileDate{2003/06/22}

% To identify the package when it will be loaded
\message{‘pst-xxxx’ v\FileVersion, \FileDate\space
                (FirstName LastName)}

% To allow other packages to test if this one
% was already loaded
\csname PstXxxxLoaded\endcsname
\let\PstXxxxLoaded\endinput

% Require of course the PSTricks package,
% and perhaps other ones
\ifx\PSTricksLoaded\endinput\else\input{pstricks}\fi
%\ifx\PSTnodesLoaded\endinput\else\input{pst-node}\fi
%\ifx\MultidoLoaded\endinput\else\input{multido}\fi

% David Carlisle interface to the ‘keyval’ package
```

---

[51] For 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.

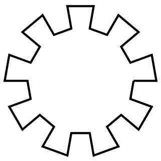
```
\input{pst-key}

% Allow to use the @ character for all internal macros
\edef\PstAtCode{\the\catcode‘\@}
\catcode‘\@=11\relax

% Definition of the specific parameters
% ============================

% A parameter containing an integer value
\define@key{psset}{Integer}{\pst@getint{#1}{\PstXxxx@Integer}}

% A parameter containing a real value
\define@key{psset}{Real}{\pst@checknum{#1}{\PstXxxx@Real}}

% A parameter containing a string value
\define@key{psset}{String}{\edef\PstXxxx@String{#1}}

% A parameter containing a boolean value
\newif\ifPstXxxx@Boolean
\define@key{psset}{Boolean}[true]{%
\@nameuse{PstXxxx@Boolean#1}}

% Other parameters
% ...

% Defaults values for these parameters
% ==============================
\setkeys{psset}{Integer=10,Real=123.456,String=,Boolean=false}

% The macro \PstXxxx, with or without optional parameters
% ============================================
\def\PstXxxx{\def\pst@par{}\pst@object{PstXxxx}}

\def\PstXxxx{{% Two braces, because parameter
              % assignments must be local
\use@par% Assignment of local parameters
% The code itself, which can of course include calls
% to auxiliary macros
% ...
}}

% Restore the definition of the @ character
\catcode‘\@=\PstAtCode\relax

\endinput
```

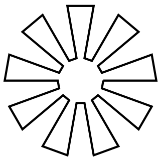
64 Examples

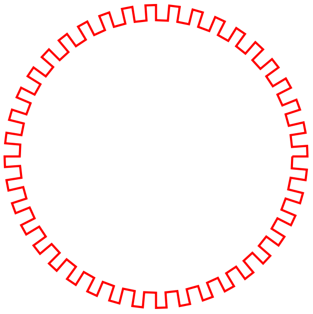
This section gives five examples which illustrate the major features of the ‘pst-key’ package and show how to use it to write high-level macros and packages. These examples are explained in step by step detail, trying to illustrate many different techniques which can be put to use by developers.

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.



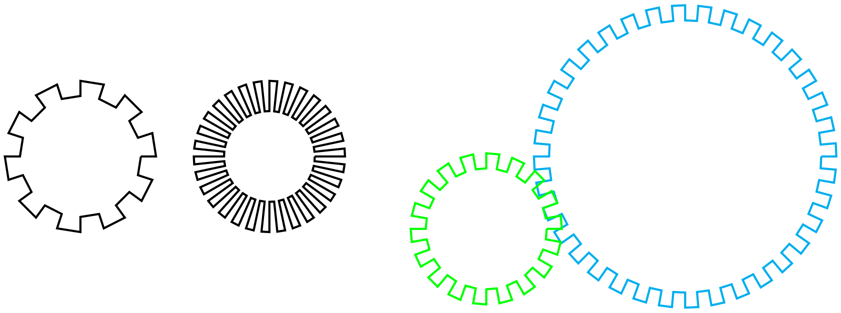




```
\SpecialCoor

\def\PstCoggedWheel#1#2{%
% #1 = number of teeth (it must be a divisor of 360!)
% #2 = height of the teeth, between 0 and 1 unit
\pst@cntc=360
\divide\pst@cntc by #1
\pst@cntd=\pst@cntc
\divide\pst@cntd\tw@
\pst@dimd=\psunit
\pssetlength{\pst@dimc}{#2}%
\advance\pst@dimd-\pst@dimc
\pscustom{%
  \moveto(\pst@dimd;0)
  \multido{\iAngleA=\z@+\pst@cntc,
           \iAngleB=\pst@cntd+\pst@cntc,
           \iAngleC=\pst@cntc+\pst@cntc}{#1}{%
    \lineto(1;\iAngleA)
    \lineto(1;\iAngleB)
    \lineto(\pst@dimd;\iAngleB)
    \lineto(\pst@dimd;\iAngleC)}}}

\rput(-2,4){\PstCoggedWheel{9}{0.3}}
\rput(-2,1){\PstCoggedWheel{9}{7mm}}
\rput(-2,-3){\psset{unit=2,linecolor=red}\PstCoggedWheel{40}{0.1}}
```

```
% “TeethNumber”: number of teeth (it must be a divisor of 360!)
\define@key{psset}{TeethNumber}{%
\pst@getint{#1}{\PstCoggedWheel@TeethNumber}}

% “TeethHeight”: height of the teeth, in current unit
\define@key{psset}{TeethHeight}{%
\edef\PstCoggedWheel@TeethHeight{#1}}

% Default parameters values
\setkeys{psset}{TeethNumber=10,TeethHeight=0.2}

% Main macro for “cogged wheel” object
```

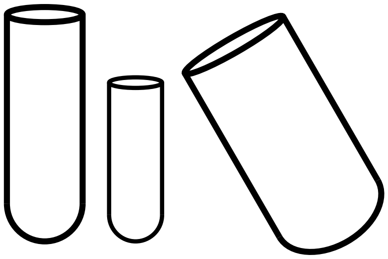
```
\def\PstCoggedWheel{\def\pst@par{}\pst@object{PstCoggedWheel}}

\def\PstCoggedWheel@i{{%
\use@par% Assignment of local parameters
\pst@cntc=360
\divide\pst@cntc\PstCoggedWheel@TeethNumber
% Test to verify if the number of teeth is valid
\pst@cnth=\pst@cntc
\multiply\pst@cnth\PstCoggedWheel@TeethNumber
\ifnum\pst@cnth=360
\else
  \@pstrickserr{TeethNumber (\PstCoggedWheel@TeethNumber)
    must be a divisor of 360! Results will be inaccurate.}{\@ehpb}%
\fi
\pst@cntd=\pst@cntc
\divide\pst@cntd\tw@
\pst@dimd=\psunit
\pssetlength{\pst@dimc}{\PstCoggedWheel@TeethHeight}%
\advance\pst@dimd-\pst@dimc
\pscustom{%
    \moveto(\pst@dimd;0)
    \multido{\iAngleA=\z@+\pst@cntc,\iAngleB=\pst@cntd+\pst@cntc,
                  \iAngleC=\pst@cntc+\pst@cntc}%
            {\PstCoggedWheel@TeethNumber}{%
        \lineto(1;\iAngleA)
        \lineto(1;\iAngleB)
        \lineto(\pst@dimd;\iAngleB)
        \lineto(\pst@dimd;\iAngleC)}}}}

\rput(-4,0){\PstCoggedWheel}
\rput(-1.5,0){\PstCoggedWheel[TeethNumber=30,TeethHeight=4mm]}
\rput(4,0){%
    \PstCoggedWheel[unit=2,linecolor=cyan,
                                TeethNumber=36,TeethHeight=0.1]%
    \rput(2.8;200){%
        \PstCoggedWheel[linecolor=green,TeethNumber=18]}}
```



64.2 Advanced example: test tube

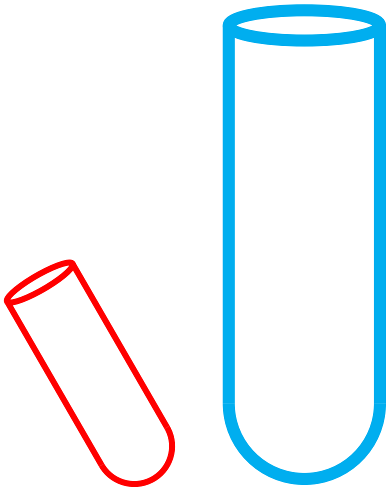
This is a complex example which defines a high level macro to draw a test tube, as used in chemistry.[52]

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.



```
\def\PstTestTube{{% Two braces to limit the scope of the
                  % next settings inside the macro only
\psset{dimen=middle,linewidth=0.08}%
\psline(-0.5,3)(-0.5,0.5)
\psellipticarc(0,0.5)(0.5,0.5){180}{0}
\psline(0.5,0.5)(0.5,3)
\psellipse(0,3)(0.5,0.1)}}

\rput(-4,0){\PstTestTube}
\rput(-2.8,0){\psset{unit=0.7}\PstTestTube}
\rput{30}(0,0){\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.



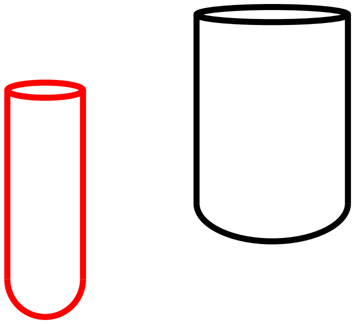
```
% Macro for tube (\PstTestTube last macro)
\def\PstLaboTestTube@Tube{{%
\psset{linewidth=0.08}%
\psline(-0.5,3)(-0.5,0.5)
\psellipticarc(0,0.5)(0.5,0.5){180}{0}
\psline(0.5,0.5)(0.5,3)
\psellipse(0,3)(0.5,0.1)}}

% Main macro for “test tube” object
\def\PstLaboTestTube{%
\def\pst@par{}\pst@object{PstLaboTestTube}}

\def\PstLaboTestTube@i{{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\PstLaboTestTube@Tube}}

\rput{30}(-3,0){\PstLaboTestTube[linecolor=red]}
\rput(-1,0){\PstLaboTestTube[unit=2,linecolor=cyan]}
```

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

---

[52] This 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){\PstLab
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.



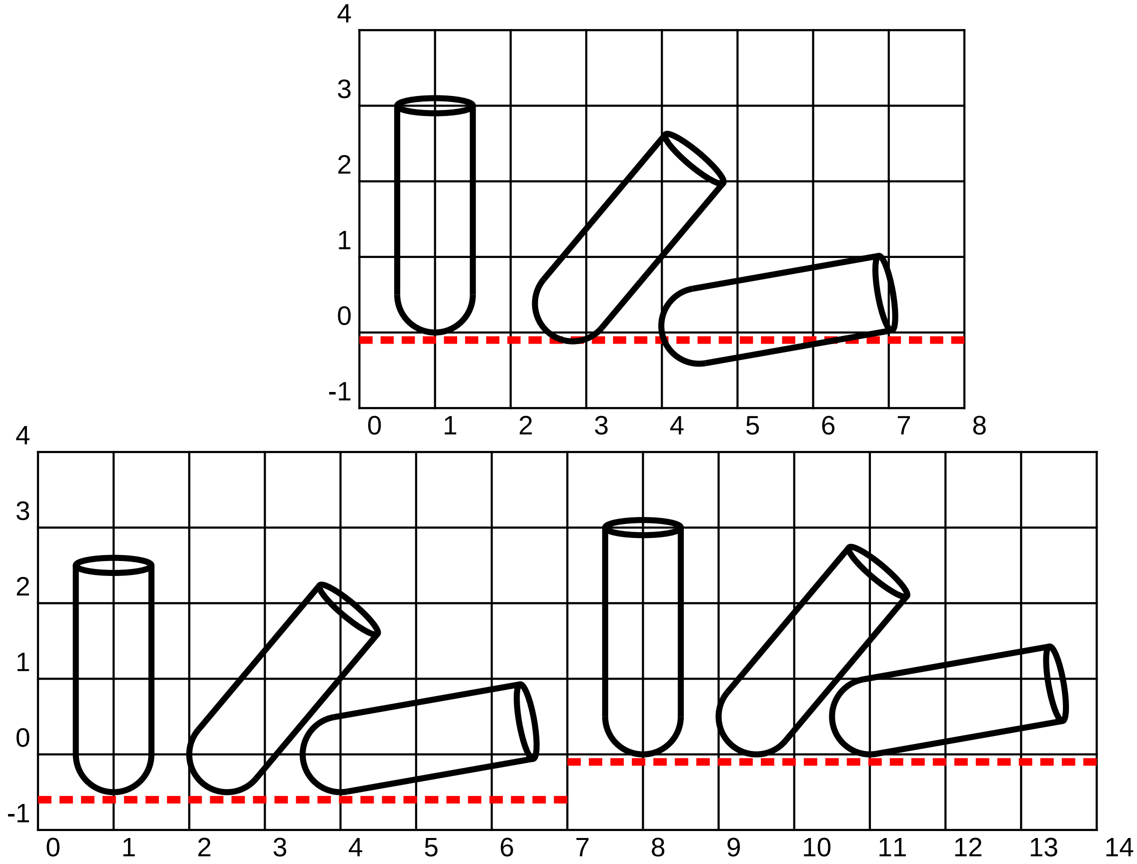
```
\def\PstLaboTestTube@i{%
% Test for optional coordinate (default = (0,0))
\@ifnextchar({\PstLaboTestTube@ii}%
               {\PstLaboTestTube@ii(\z@,\z@)}}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){\PstLaboTestTube@Tube}}}

\PstLaboTestTube[linecolor=red](-3,0)
\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 it basis will be at the (0,0) coordinate.)

```
\def\PstTestTubeA{{%
\psset{dimen=middle,linewidth=0.08}%
\psline(-0.5,3)(-0.5,0.5)
\psellipticarc(0,0.5)(0.5,0.5){180}{0}
\psline(0.5,0.5)(0.5,3)
\psellipse(0,3)(0.5,0.1)}}

\def\PstTestTubeB{{%
\psset{dimen=middle,linewidth=0.08,origin={0,0.5}}%
\psline(-0.5,3)(-0.5,0.5)
\psellipticarc(0,0.5)(0.5,0.5){180}{0}
\psline(0.5,0.5)(0.5,3)
\psellipse(0,3)(0.5,0.1)}}

\rput(1,0){\PstTestTubeA}
```

```
\rput{-40}(2.5,0){\PstTestTubeA}
\rput{-80}(4,0){\PstTestTubeA}

\rput(1,0){\PstTestTubeB}
\rput{-40}(2.5,0){\PstTestTubeB}
\rput{-80}(4,0){\PstTestTubeB}
\rput(8,0.5){\PstTestTubeB}
\rput{-40}(9.5,0.5){\PstTestTubeB}
\rput{-80}(11,0.5){\PstTestTubeB}
```

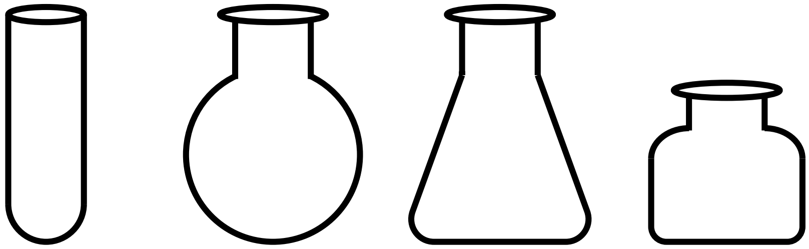


The definitions of the six macros \PstLaboTestTube@Height (the height of the tube), \PstLaboTestTube@HeightTop (the height of the bottom part of the tube above the rotation point), \PstLaboTestTube@HeightBottom (the height of the bottom part of the tube below the rotation point), \PstLaboTestTube@WidthBottomHalf (half the width of the bottom part), \PstLaboTestTube@WidthTopHalf (half the width of the top part) and \PstLaboTestTube@W (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 it value before invoking the macro defining the kind of tube.

```
% Form parameter
\define@key{psset}{Form}{%
\def\PstLaboTestTube@Tube{\@nameuse{PstLaboTestTube@#1}}%
\def\PstLaboTestTube@HeightBottom{%
\@nameuse{PstLaboTestTube@#1@HeightBottom}}}

\setkeys{psset}{Form=straight}% Default value

% Macro for straight tube (was \PstLaboTestTube@Tube)
\def\PstLaboTestTube@straight{{%
\gdef\PstLaboTestTube@Height{3}%
\gdef\PstLaboTestTube@HeightTop{2.5}%
\gdef\PstLaboTestTube@WidthBottomHalf{0.5}%
\gdef\PstLaboTestTube@WidthTopHalf{0.5}%
\gdef\PstLaboTestTube@WidthNeckHalf{0.5}%
\psset{dimen=middle,linewidth=0.08,
      origin={0,\PstLaboTestTube@straight@HeightBottom}}%
\psline(-0.5,3)(-0.5,0.5)
\psellipticarc(0,0.5)(0.5,0.5){180}{0}
\psline(0.5,0.5)(0.5,3)
\psellipse(0,3)(0.5,0.1)}}
\def\PstLaboTestTube@straight@HeightBottom{0.5}

% Macro for round bottom flask
\def\PstLaboTestTube@roundbottom{{%
\gdef\PstLaboTestTube@Height{3}%
\gdef\PstLaboTestTube@HeightTop{1.8}%
\gdef\PstLaboTestTube@WidthBottomHalf{1.15}%
\gdef\PstLaboTestTube@WidthTopHalf{0.5}%
\gdef\PstLaboTestTube@WidthNeckHalf{0.7}%
\psset{dimen=middle,linewidth=0.08,%
      origin={0,\PstLaboTestTube@roundbottom@HeightBottom}}%
\psline(-0.5,2.95)(-0.5,2.15)%
\psellipticarc(0,1.15)(1.15,1.15){115}{65}%
\psline(0.5,2.15)(0.5,2.95)%
\psellipse(0,3)(0.7,0.1)}}
\def\PstLaboTestTube@roundbottom@HeightBottom{1.2}

% Macro for erlenmeyer flask
\def\PstLaboTestTube@erlenmeyer{{%
\gdef\PstLaboTestTube@Height{3}%
\gdef\PstLaboTestTube@HeightTop{1.5}%
\gdef\PstLaboTestTube@WidthBottomHalf{1.3}%
\gdef\PstLaboTestTube@WidthTopHalf{0.5}%
\gdef\PstLaboTestTube@WidthNeckHalf{0.7}%
\psset{dimen=middle,linewidth=0.08,%
      origin={0,\PstLaboTestTube@erlenmeyer@HeightBottom}}%
% Last point required to be able to fill the region correctly
\psline(-0.5,2.9)(-0.5,2.2)(-0.5,2.25)
\psline[linearc=0.3](-0.5,2.2)(-1.3,0)(1.3,0)(0.5,2.2)
\psline(0.5,2.2)(0.5,2.9)
\psellipse(0,3)(0.7,0.1)}}
```

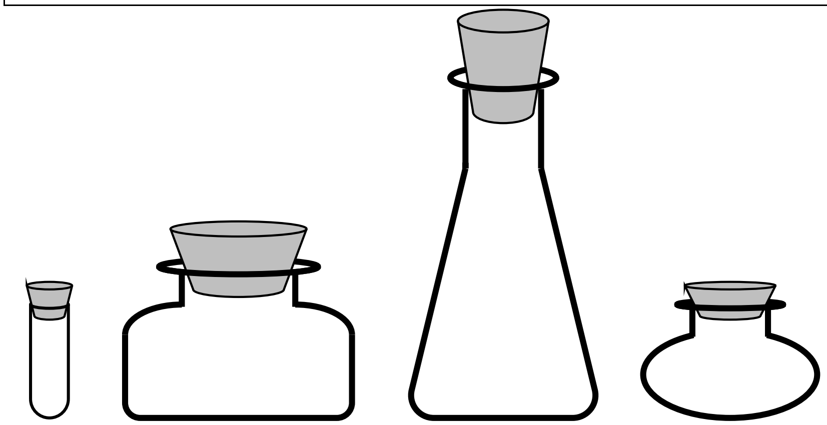
```
\def\PstLaboTestTube@erlenmeyer@HeightBottom{1.5}

% Macro for small flask
\def\PstLaboTestTube@small{{%
\gdef\PstLaboTestTube@Height{2}%
\gdef\PstLaboTestTube@HeightTop{0.8}%
\gdef\PstLaboTestTube@WidthBottomHalf{1}%
\gdef\PstLaboTestTube@WidthTopHalf{0.5}%
\gdef\PstLaboTestTube@WidthNeckHalf{0.7}%
\psset{dimen=middle,linewidth=0.08,%
         origin={0,\PstLaboTestTube@small@HeightBottom}}%
\psline(-0.5,1.9)(-0.5,1.47)
\psellipticarc(-0.5,1.1)(0.5,0.4){90}{180}
\psline[linearc=0.2](-1,1.1)(-1,0)(1,0)(1,1.1)
\psellipticarc(0.5,1.1)(0.5,0.4){0}{90}
\psline(0.5,1.47)(0.5,1.9)
\psellipse(0,2)(0.7,0.1)}}
\def\PstLaboTestTube@small@HeightBottom{1.2}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
   \rput(0,\PstLaboTestTube@HeightBottom){%
      \PstLaboTestTube@Tube}}}}

\PstLaboTestTube(-4.5,0)
\PstLaboTestTube[Form=roundbottom](-1.5,0)
\PstLaboTestTube[Form=erlenmeyer](1.5,0)
\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.

```
% Stopper parameter
\newif\ifPstLaboTestTube@Stopper
\define@key{psset}{Stopper}[true]{%
\@nameuse{PstLaboTestTube@Stopper#1}}

\setkeys{psset}{Stopper=false}% Default value
```

```

% Macro for stopper (depends on the height of the tube
% and on the width of the neck)
\def\PstLaboTestTube@Stopper{%
\rput(0,\PstLaboTestTube@HeightTop){%
  \pscustom[fillstyle=solid,fillcolor=lightgray]{%
    \psellipticarc(0,0.5)(0.6,0.1){0}{180}
    \psline(-0.6,0.5)(-0.4,-0.3)
    \psellipticarc(0,-0.3)(0.4,0.1){180}{0}
    \psline(0.4,-0.3)(0.6,0.5)}
  \psellipticarc(0,0.5)(0.6,0.1){180}{0}
  \psellipticarc[linewidth=0.08]
    (\PstLaboTestTube@WidthNeckHalf,0.1){180}{0}}}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
  \rput(0,\PstLaboTestTube@HeightBottom){%
    \PstLaboTestTube@Tube
    \ifPstLaboTestTube@Stopper
      \PstLaboTestTube@Stopper
    \fi}}}}

\psset{Stopper=true}
\rput(-4.5,0){\PstLaboTestTube[unit=0.5]}
\rput(-2,0){\PstLaboTestTube[xunit=1.5,Form=small]}
\rput(1.5,0){\PstLaboTestTube[yunit=1.5,Form=erlenmeyer]}
\rput(4.5,0){\PstLaboTestTube[yunit=0.5,Form=roundbottom]}
```

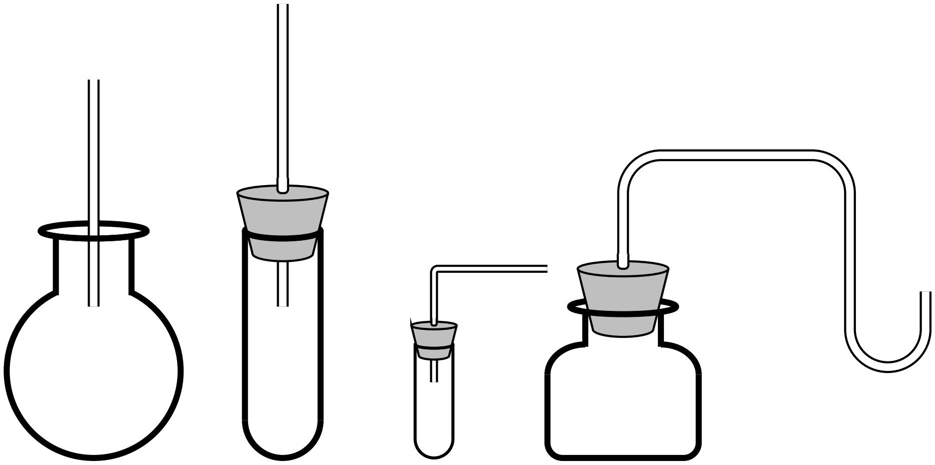


Now, we will add the possibility of having a small escape tube, straight or bent, through the stopper. For this we define a *string* parameter, which can take here the values straight, bent or triplebent (it will be easy to add new ones of any kind), and a *length* parameter (but defined as a macro, 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.

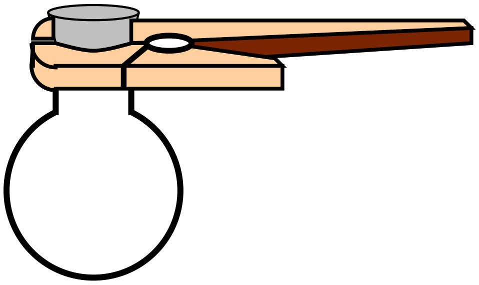
```
% Escape tube type parameter
\define@key{psset}{EscapeTube}{%
\def\PstLaboTestTube@EscapeTube{%
\@nameuse{PstLaboTestTube@EscapeTube@#1}}}

% Escape tube length parameter
\define@key{psset}{EscapeTubeLength}{%
\edef\PstLaboTestTube@EscapeTubeLength{#1}}

\setkeys{psset}{EscapeTube=,EscapeTubeLength=3}% Defaults

% Macro for no escape tube
\let\PstLaboTestTube@EscapeTube@\relax

% Macro for straight escape tube
\def\PstLaboTestTube@EscapeTube@straight{{%
\pssetlength{\pst@dimd}{\PstLaboTestTube@HeightTop}%
\psaddtolength{\pst@dimd}{\m@ne}%
\rput(0,\pst@dimd){%
  \psline[doubleline=true,doublesep=0.1]
       (0,\PstLaboTestTube@EscapeTubeLength)}
\PstLaboTestTube@EscapeTube@AdjustmentsStopper}}

% Macro for bent escape tube
\def\PstLaboTestTube@EscapeTube@bent{{%
\pssetlength{\pst@dimd}{\PstLaboTestTube@HeightTop}%
\psaddtolength{\pst@dimd}{\m@ne}%
\rput(0,\pst@dimd){%
  \psline[doubleline=true,doublesep=0.1,linearc=0.1]
       (0,0)(0,3)(\PstLaboTestTube@EscapeTubeLength,3)}
\PstLaboTestTube@EscapeTube@AdjustmentsStopper}}

% Macro for triple bent escape tube
\def\PstLaboTestTube@EscapeTube@triplebent{{%
\pssetlength{\pst@dimc}{\PstLaboTestTube@HeightTop}%
\psaddtolength{\pst@dimc}{\tw@}%
\pssetlength{\pst@dimd}{\PstLaboTestTube@EscapeTubeLength}%
\psaddtolength{\pst@dimd}{\@ne}%
\psline[doubleline=true,doublesep=0.1,linearc=0.5]
     (0,0.5)(0,\pst@dimc)
     (\PstLaboTestTube@EscapeTubeLength,\pst@dimc)
     (\PstLaboTestTube@EscapeTubeLength,0)
     (\pst@dimd,0)(\pst@dimd,1)
\PstLaboTestTube@EscapeTube@AdjustmentsStopper}}

% If there is no stopper, redraw of the external neck
\def\PstLaboTestTube@EscapeTube@AdjustmentsStopper{%
\ifPstLaboTestTube@Stopper
\else
  \psellipticarc[linewidth=0.08](0,\PstLaboTestTube@HeightTop)
    (\PstLaboTestTube@WidthNeckHalf,0.1){180}{0}
\fi}
```

```

\def\PstLaboTestTube@Stopper{%
\rput(0,\PstLaboTestTube@HeightTop){%
  \pscustom[fillstyle=solid,fillcolor=lightgray]{%
    \psellipticarc(0,0.5)(0.6,0.1){0}{180}
    \psline(-0.6,0.5)(-0.4,-0.3)
    \psellipticarc(0,-0.3)(0.4,0.1){180}{0}
    \psline(0.4,-0.3)(0.6,0.5)}
  \psellipticarc(0,0.5)(0.6,0.1){180}{0}
  \psellipticarc[linewidth=0.08]
    (\PstLaboTestTube@WidthNeckHalf,0.1){180}{0}
 \if\PstLaboTestTube@EscapeTube\relax
 \else
   \let\psfillcolor\psdoublecolor
   \pscustom[fillstyle=solid]{%
     \psline[linearc=0.05](-0.07,0.7)(-0.07,0.5)(0.07,0.5)(0.07,0.7)}
 \fi}}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
  \rput(0,\PstLaboTestTube@HeightBottom){%
    \expandafter\PstLaboTestTube@Tube
    \expandafter\PstLaboTestTube@EscapeTube
    \ifPstLaboTestTube@Stopper
      \PstLaboTestTube@Stopper
    \fi}}}}

\PstLaboTestTube[Form=roundbottom,EscapeTube=straight](-5,0)
\PstLaboTestTube[Stopper=true,EscapeTube=straight,
  EscapeTubeLength=4](-2.5,0)
\PstLaboTestTube[unit=0.5,Stopper=true,EscapeTube=bent](-1,0)
\PstLaboTestTube[Form=small,Stopper=true,
  EscapeTube=triplebent](2,0)
```



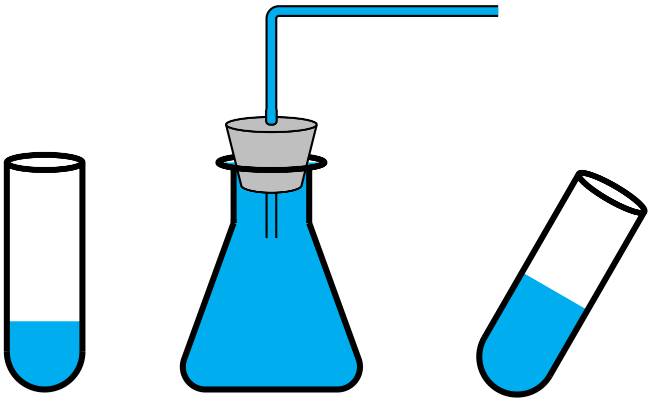
We can still easily add various objects to enrich the possibilities.

```
% Clip parameter
\newif\ifPstLaboTestTube@Clip
\define@key{psset}{Clip}[true]{\@nameuse{PstLaboTestTube@Clip#1}}

% Macro for clip
\def\PstLaboTestTube@Clip{%
% We put the clip at vertical position 0.7*height
\pst@dimh=\PstLaboTestTube@HeightTop\psyunit
\rput(-0.8,0.72\pst@dimh){%
  \psset{linewidth=0.05}%
  \pspolygon[fillstyle=solid,fillcolor=LightOrange]
           (1.3,0.9)(5.7,0.9)(5.8,0.8)(1.8,0.6)(1.3,0.6)
  \pspolygon[fillstyle=solid,fillcolor=Brown]
           (1.3,0.6)(5.8,0.8)(5.8,0.6)(2.8,0.4)(1.8,0.6)
  \pscustom[fillstyle=solid,fillcolor=LightOrange]{%
    \pscurve(1.3,0.6)(0.8,0.5)(0.3,0.6)
    \psline(0.3,0.6)(0,0.6)
    \psellipticarc(0.3,0.3)(0.32,0.32){180}{270}
    \psline(0.3,0.3)(3.3,0.3)(3.2,0.4)(1.8,0.6)(1.3,0.6)}
  \pscustom[fillstyle=solid,fillcolor=LightOrange]{%
    \psline(0,0.6)(0,0.3)
    \psellipticarc(0.3,0.3)(0.32,0.32){180}{270}
    \psline(0.3,0)(3.3,0)(3.3,0.3)(0.3,0.3)
    \psellipticarcn(0.3,0.6)(0.32,0.32){-90}{180}}
  \pswedge[fillstyle=solid,fillcolor=LightOrange]
          (0.27,0.66){0.27}{90}{180}
  \psellipse[linewidth=1.5\pslinewidth,fillstyle=solid]
          (1.8,0.6)(0.3,0.1)
  \psline[linewidth=1.5\pslinewidth](1.5,0.55)(1.2,0.3)(1.2,0)}}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
  \rput(0,\PstLaboTestTube@HeightBottom){%
    \expandafter\PstLaboTestTube@Tube
    \expandafter\PstLaboTestTube@EscapeTube
    \ifPstLaboTestTube@Stopper
      \PstLaboTestTube@Stopper
    \fi
    \ifPstLaboTestTube@Clip
      \PstLaboTestTube@Clip
    \fi}}}}

\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.

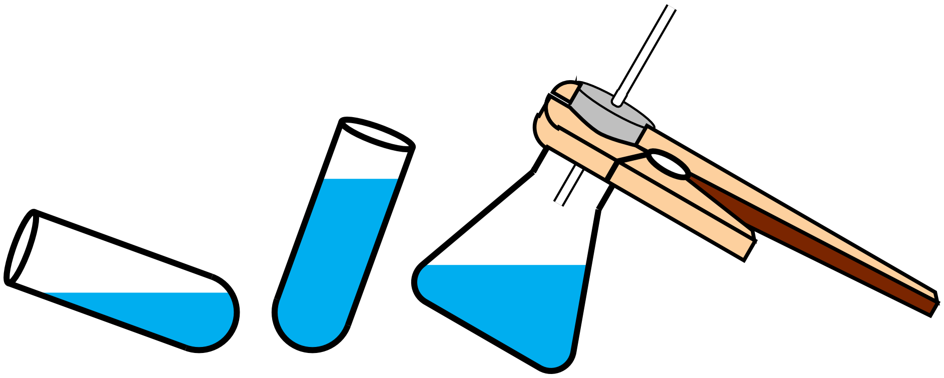
```
\SpecialCoor

% LiquidLevel parameter
\define@key{psset}{LiquidLevel}{%
\pst@checknum{#1}{\PstLaboTestTube@LiquidLevel}}

\setkeys{psset}{LiquidLevel=0}% Default value

% Macro for liquid
\def\PstLaboTestTube@Liquid#1{%
\psframe[linestyle=none,fillstyle=solid,fillcolor=cyan]
       (-4,-2)(! 4 \PstLaboTestTube@Height\space #1 mul)}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
  \psclip{\pscustom[linestyle=none]{%
         \expandafter\PstLaboTestTube@Tube}}
    \PstLaboTestTube@Liquid{\PstLaboTestTube@LiquidLevel}
  \endpsclip
  \rput(0,\PstLaboTestTube@HeightBottom){%
      % ...
    \fi}}}}

\PstLaboTestTube[LiquidLevel=0.3](-3,0)
\PstLaboTestTube[Form=erlenmeyer,Stopper=true,
  EscapeTube=bent,LiquidLevel=1,doublecolor=cyan]
\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.

```
% Angle parameter
\define@key{psset}{Angle}{%
\pst@getangle{#1}{\PstLaboTestTube@Angle}}

\setkeys{psset}{Angle=0}% Default value

% Macro for liquid (#1 = angle, #2 = liquid level)
\def\PstLaboTestTube@Liquid#1#2{%
\pst@Verb{%
  /xA {\PstLaboTestTube@WidthBottomHalf\space
      #1 0 gt {neg} if} def
  /LiquidHeight {#2 \PstLaboTestTube@Height\space mul
              #2 \PstLaboTestTube@HeightBottom\space mul
              #1 sin abs mul sub} def
  /xA' {xA #1 cos mul
       \PstLaboTestTube@Height\space #1 sin mul sub} def
  #1 0 gt {/Xinf xA' def /Xsup 2 def}
          {/Xinf -2 def /Xsup xA' def} ifelse}%
\psframe[linestyle=none,fillstyle=solid,fillcolor=cyan]
       (! Xinf -2)(! Xsup LiquidHeight)}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){{%
  \psclip{\rput{\PstLaboTestTube@Angle}
           (0,\PstLaboTestTube@HeightBottom){%
         \pscustom[linestyle=none]{%
           \translate(0,-\PstLaboTestTube@HeightBottom)
           \expandafter\PstLaboTestTube@Tube}}}
    \PstLaboTestTube@Liquid{\PstLaboTestTube@Angle}
                         {\PstLaboTestTube@LiquidLevel}%
  \endpsclip
  \rput{\PstLaboTestTube@Angle}
```

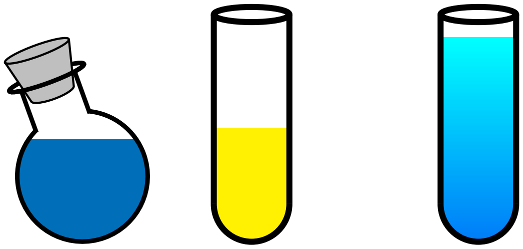
```
        (0,\PstLaboTestTube@HeightBottom){%
      \expandafter\PstLaboTestTube@Tube
    % ...
      \fi}}}}}

\PstLaboTestTube[Angle=70,LiquidLevel=0.3](-4,0)
\PstLaboTestTube[Angle=-20,LiquidLevel=0.8](-2.5,0)
\PstLaboTestTube[Angle=-30,Form=erlenmeyer,Stopper=true,
    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.

```
% LiquidType parameter
\define@key{psset}{LiquidType}{%
\def\PstLaboTestTube@LiquidType{#1}}

% Default value (first is the parameter name,
% second is the aspect name!)
\setkeys{psset}{LiquidType=LiquidType}
\newpsstyle{LiquidType}{%
    linestyle=none,fillstyle=solid,fillcolor=cyan}

% Macro for liquid (#1 = angle, #2 = liquid level, #3 = style)
\def\PstLaboTestTube@Liquid#1#2#3{%
\pst@Verb{%
% ...
                {/Xinf -2 def /Xsup xA' def} ifelse}%
\psframe[style=#3](! Xinf -2)(! Xsup LiquidHeight)}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
    \psclip{\rput{\PstLaboTestTube@Angle}
                  (0,\PstLaboTestTube@HeightBottom){%
                \pscustom[linestyle=none]{%
                  \translate(0,-\PstLaboTestTube@HeightBottom)
                  \expandafter\PstLaboTestTube@Tube}}}
      \PstLaboTestTube@Liquid{\PstLaboTestTube@Angle}
```

```
                                  {\PstLaboTestTube@LiquidLevel}
                                  {\PstLaboTestTube@LiquidType}
  \endpsclip
  % ...
    \fi}}}}

\newpsstyle{cobalt}{%
  linestyle=none,fillstyle=solid,fillcolor=NavyBlue}
\newpsstyle{diffusion}{%
  linestyle=none,fillstyle=gradient,gradmidpoint=0}
\newpsstyle{oil}{linestyle=none,fillstyle=solid,fillcolor=yellow}

\PstLaboTestTube[unit=0.75,Form=roundbottom,Stopper=true,
  Angle=20,LiquidType=cobalt,LiquidLevel=0.7](-3,0)
\PstLaboTestTube[LiquidType=oil,LiquidLevel=0.5]
\PstLaboTestTube[LiquidType=diffusion,LiquidLevel=0.9](3,0)
```

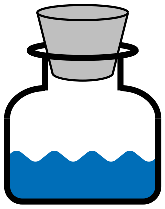


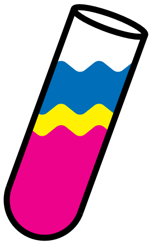
If we want to manage several liquids (here up to 3), we have only to loop on them. Note that we use a number inside the macro names. We could have use the `\@namedef` macro for that, but here we want to call the `\pst@checknum` macro to verify if the value is a well formatted number. `pst@checknum` does not allow numeric characters inside macro names, so we use use a variation on it, the `\@nameedef` macro, which does allow numeric characters inside macro names.

```
% LiquidLevel parameters
\define@key{psset}{LiquidLevel1}{%
\pst@checknum{#1}{\pst@temph}%
\expandafter\edef\csname PstLaboTestTube@LiquidLevel1%
                   \endcsname{\pst@temph}}
\define@key{psset}{LiquidLevel2}{%
\pst@checknum{#1}{\pst@temph}%
\expandafter\edef\csname PstLaboTestTube@LiquidLevel2%
                   \endcsname{\pst@temph}}
\define@key{psset}{LiquidLevel3}{%
\pst@checknum{#1}{\pst@temph}%
\expandafter\edef\csname PstLaboTestTube@LiquidLevel3%
                   \endcsname{\pst@temph}}

% LiquidType parameters
\define@key{psset}{LiquidType1}{%
\@namedef{PstLaboTestTube@LiquidType1}{#1}}
\define@key{psset}{LiquidType2}{%
\@namedef{PstLaboTestTube@LiquidType2}{#1}}
\define@key{psset}{LiquidType3}{%
\@namedef{PstLaboTestTube@LiquidType3}{#1}}

% Default values
\setkeys{psset}{LiquidType1=LiquidType1,LiquidLevel1=0,
  LiquidType2=,LiquidLevel2=0,LiquidType3=,LiquidLevel3=0}

% Default style
\newpsstyle{LiquidType1}{%
  linestyle=none,fillstyle=solid,fillcolor=cyan}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
  \psclip{\rput{\PstLaboTestTube@Angle}
           (0,\PstLaboTestTube@HeightBottom){%
         \pscustom[linestyle=none]{%
           \translate(0,-\PstLaboTestTube@HeightBottom)
           \expandafter\PstLaboTestTube@Tube}}}
   \multido{\iLiquid=\@ne+\@ne}{\thr@@}{%
    \ifdim\@nameuse{PstLaboTestTube@LiquidLevel\iLiquid}\p@>\z@
      \PstLaboTestTube@Liquid{\PstLaboTestTube@Angle}
        {\@nameuse{PstLaboTestTube@LiquidLevel\iLiquid}}
        {\@nameuse{PstLaboTestTube@LiquidType\iLiquid}}
    \fi}
  \endpsclip
  % ...
    \fi}}}}

\newpsstyle{vinegar}{linestyle=none,fillstyle=solid,fillcolor=magenta}

\PstLaboTestTube[Form=roundbottom,Stopper=true,
  LiquidType1=oil,LiquidLevel1=0.6,
  LiquidType2=vinegar,LiquidLevel2=0.2](-3,0)
\PstLaboTestTube[Angle=-20,LiquidType1=cobalt,LiquidLevel1=0.8,
  LiquidType2=oil,LiquidLevel2=0.7,
  LiquidType3=vinegar,LiquidLevel3=0.5](3,0)
```

We may want to change the form of the liquid surface. This is easy to do.

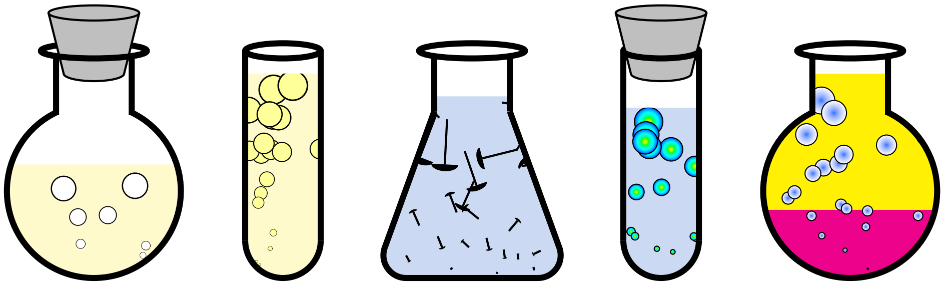
```
% Liquid surface parameter
\define@key{psset}{LiquidSurface}{%
\def\PstLaboTestTube@LiquidSurface{%
\@nameuse{PstLaboTestTube@LiquidSurface@#1}}}

% Default value
\setkeys{psset}{LiquidSurface=straight}

% Macro for liquid surface (straight)
% (#1 = angle, #2 = liquid level, #3 = style)
\def\PstLaboTestTube@LiquidSurface@straight#1#2#3{%
\pst@Verb{%
% ...
          {/Xinf -2 def /Xsup xA' def} ifelse}%
\psframe[style=#3](! Xinf -2)(! Xsup LiquidHeight)}

% Macro for liquid surface (wavy)
% (#1 = angle, #2 = liquid level, #3 = style)
\def\PstLaboTestTube@LiquidSurface@wavy#1#2#3{%
\pst@Verb{%
% ...
          {/Xinf -2 def /Xsup xA' def} ifelse}%
\pscustom[style=#3]{%
\psline(! Xinf LiquidHeight)(!Xinf -2)(! Xsup -2)(! Xsup LiquidHeight)
\pszigzag[linearc=0.1,coilwidth=0.2,coilheight=2.5]
        (! Xinf LiquidHeight)}}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
  \psclip{\rput{\PstLaboTestTube@Angle}
           (0,\PstLaboTestTube@HeightBottom){%
         \pscustom[linestyle=none]{%
           \translate(0,-\PstLaboTestTube@HeightBottom)
           \expandafter\PstLaboTestTube@Tube}}}
   \multido{\iLiquid=\@ne+\@ne}{\thr@@}{%
    \ifdim\@nameuse{PstLaboTestTube@LiquidLevel\iLiquid}\p@>\z@
      \expandafter\PstLaboTestTube@LiquidSurface%
        {\PstLaboTestTube@Angle}
        {\@nameuse{PstLaboTestTube@LiquidLevel\iLiquid}}
        {\@nameuse{PstLaboTestTube@LiquidType\iLiquid}}
    \fi}
  \endpsclip
  % ...
    \fi}}}}

\psset{LiquidSurface=wavy}
\PstLaboTestTube[Form=small,Stopper=true,
  LiquidType1=cobalt,LiquidLevel1=0.3](-3,0)
\PstLaboTestTube[Angle=-20,LiquidType1=cobalt,LiquidLevel1=0.8,
  LiquidType2=oil,LiquidLevel2=0.6,
  LiquidType3=vinegar,LiquidLevel3=0.5](3,0)
```





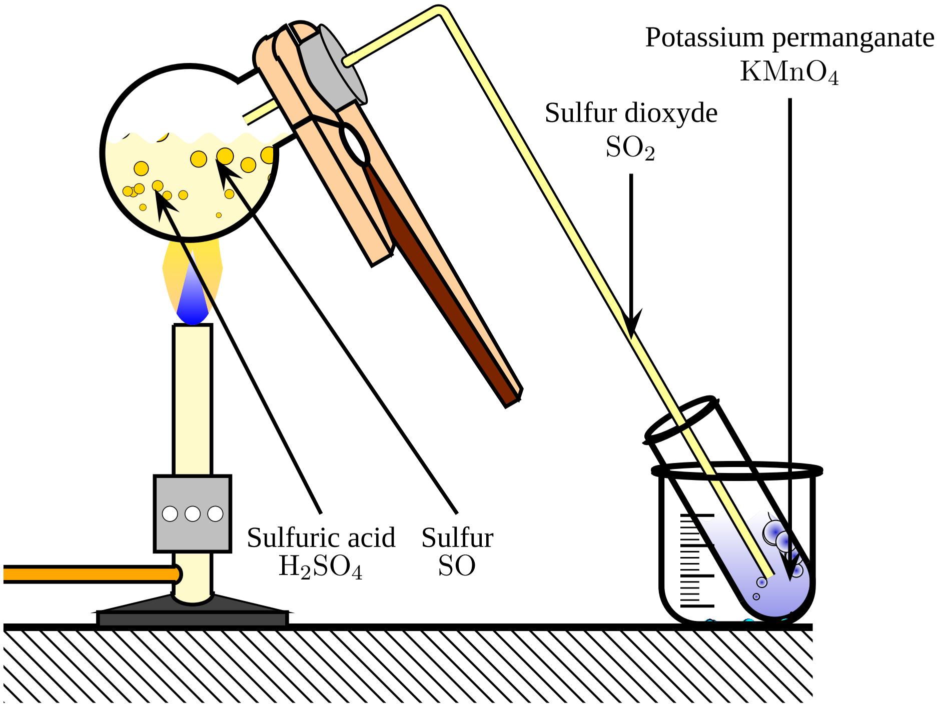
We can also add various substances inside the tubes. Bubbles and nails are defined below, in random sizes and at random positions.

```
% Substance parameter
\define@key{psset}{Substance}{%
\def\PstLaboTestTube@Substance{%
\@nameuse{PstLaboTestTube@Substance@#1}}}

% Substance number parameter
\define@key{psset}{SubstanceNumber}{%
\pst@getint{#1}{\PstLaboTestTube@SubstanceNumber}}

% Substance style parameter
\define@key{psset}{SubstanceStyle}{%
\edef\PstLaboTestTube@SubstanceStyle{#1}}

% Default values
\setkeys{psset}{Substance=,SubstanceNumber=10,SubstanceStyle=}

% Macro to draw bubbles of random sizes at random positions
\def\PstLaboTestTube@Substance@bubbles{%
\PstLaboTestTube@Substance@i{%
  \psdot[dotstyle=o,dotscale=\pointless\pst@dimh]
    (! \pst@number{\pst@dimc} \pst@number{\pst@dimd}
        \@nameuse{PstLaboTestTube@LiquidLevel1} mul)}}

% Macro to draw colored bubbles of random sizes at random pos
\def\PstLaboTestTube@Substance@coloredbubbles{%
\PstLaboTestTube@Substance@i{%
  \pscircle[linewidth=0.5\pslinewidth,fillstyle=ccslope,
         runit=\pointless\pst@dimh]
    (! \pst@number{\pst@dimc} \pst@number{\pst@dimd}
        \@nameuse{PstLaboTestTube@LiquidLevel1} mul){0.06}}}

% Macro to draw nails of random sizes at random positions
\def\PstLaboTestTube@Substance@nails{%
\PstLaboTestTube@Substance@i{%
  \rput{\the\pst@cnth}%
      (! \pst@number{\pst@dimc} \pst@number{\pst@dimd}
        \@nameuse{PstLaboTestTube@LiquidLevel1} mul){%
    \psset{unit=\pointless\pst@dimh}%
    \psline(0,0.2)
    \psellipticarc*(0,0.2)(0.06,0.03){0}{180}}}}

% Macro to draw thin objects of random sizes at random pos.
\def\PstLaboTestTube@Substance@i#1{% #1 = macro of the object
\multido{\iSubstance=\@ne+\@ne}
      {\PstLaboTestTube@SubstanceNumber}{%
  \setrandim{\pst@dimc}{-\PstLaboTestTube@WidthBottomHalf\p@}
    {\PstLaboTestTube@WidthBottomHalf\p@}% For position
  % For position and size
  \setrandim{\pst@dimd}{\z@}{\PstLaboTestTube@Height\p@}%
  \setrannum{\pst@cnth}{\z@}{360}% For angle
  % The size increase proportionally to the vertical position
  \pst@dimh=\pst@dimd
  % And this size must depend of the unit
  \pst@dimg=0.04\psunit
  \pst@dimh=\pointless\pst@dimg\pst@dimh
  \ifx\PstLaboTestTube@SubstanceStyle\@empty
  \else
    \setkeys{psset}{style=\PstLaboTestTube@SubstanceStyle}%
  \fi
  #1}}

\def\PstLaboTestTube@ii(#1){{%
\setkeys{psset}{dimen=middle}%
\use@par% Assignment of local parameters
\rput(#1){%
  \psclip{\rput{\PstLaboTestTube@Angle}
           (0,\PstLaboTestTube@HeightBottom){%
         \pscustom[linestyle=none]{%
           \translate(0,-\PstLaboTestTube@HeightBottom)
           \expandafter\PstLaboTestTube@Tube}}
         % To clip the optional substance by the first liquid
         \expandafter\PstLaboTestTube@LiquidSurface%
           {\PstLaboTestTube@Angle}
           {\@nameuse{PstLaboTestTube@LiquidLevel1}}{none}}
    % ...
    \PstLaboTestTube@Substance
  \endpsclip
  % ...
    \fi}}}}

\newpsstyle{champagne}{%
  linestyle=none,fillstyle=solid,fillcolor=LemonChiffon}
\newpsstyle{water}{%
  linestyle=none,fillstyle=solid,fillcolor=LightBlue}
\newpsstyle{BubblesChampagne}{%
  fillstyle=solid,fillcolor=PaleYellow}

\PstLaboTestTube[Form=roundbottom,Stopper=true,
  LiquidType1=champagne,LiquidLevel1=0.5,
  Substance=bubbles](-5,0)
\PstLaboTestTube[LiquidType1=champagne,
  LiquidLevel1=0.9,Substance=bubbles,SubstanceNumber=20,
  SubstanceStyle=BubblesChampagne](-2.5,0)
\PstLaboTestTube[Form=erlenmeyer,LiquidType1=water,
  LiquidLevel1=0.8,Substance=nails,SubstanceNumber=30]
\PstLaboTestTube[Stopper=true,LiquidType1=water,LiquidLevel1=0.75,
  Substance=coloredbubbles,SubstanceNumber=15](2.5,0)
\PstLaboTestTube[Form=roundbottom,LiquidType1=oil,
  LiquidLevel1=0.9,LiquidType2=vinegar,LiquidLevel2=0.3,
  Substance=coloredbubbles,SubstanceNumber=25,
  slopebegin=NavyBlue,slopeend=white](5,0)
```



And we can of course compose several objects to illustrate complex scenes.

```
% Macro for becher
\def\PstLaboTestTube@becher{{%
\gdef\PstLaboTestTube@Height{2.5}%
\gdef\PstLaboTestTube@HeightTop{1.275}%
\gdef\PstLaboTestTube@WidthBottomHalf{0.55}%
\gdef\PstLaboTestTube@WidthTopHalf{0.55}%
\gdef\PstLaboTestTube@WidthNeckHalf{0.55}%
\psset{dimen=middle,linewidth=0.08,%
      origin={0,\PstLaboTestTube@becher@HeightBottom}}%
\psellipse(0,2.55)(1.1,0.1)
\psline[linearc=0.5](-1,2.5)(-1,0)(1,0)(1,2.5)
\multido{\nDiv=0.3+0.5}{4}{%
  \psline[linewidth=0.5\pslinewidth](-0.75,\nDiv)(-0.3,\nDiv)}
\multido{\nSubDiv=0.3+0.1}{16}{%
  \psline[linewidth=0.25\pslinewidth](-0.75,\nSubDiv)(-0.5,\nSubDiv)}}}
\def\PstLaboTestTube@becher@HeightBottom{1.275}

% Macro for "Bunsen burner" object
\def\PstLaboBunsenBurner{%
\def\pst@par{}\pst@object{PstLaboBunsenBurner}}

\def\PstLaboBunsenBurner@i{%
% Test for optional angle (default = 0)
\@ifnextchar\bgroup{\PstLaboBunsenBurner@ii}%
                 {\PstLaboBunsenBurner@ii{0}}}

\def\PstLaboBunsenBurner@ii#1{% #1 = angle
% Test for optional coordinate (default = (0,0))
\@ifnextchar({\PstLaboBunsenBurner@iii{#1}}%
                    {\PstLaboBunsenBurner@iii{#1}(\z@,\z@)}}

\def\PstLaboBunsenBurner@iii#1(#2){{%
% #1 = angle, #2 = coordinate
\setkeys{psset}{linewidth=0.05}%
\use@par% Assignment of local parameters
\rput{#1}(#2){%
  % Basis
  \pstriangle[fillstyle=solid,fillcolor=darkgray,linewidth=0.5\pslinewidth]
              (0,0.2)(2.5,0.3)
  \psframe[fillstyle=solid,fillcolor=darkgray](-1.25,0)(1.25,0.2)
  % Burner
  \pscustom[fillstyle=solid,fillcolor=LemonChiffon]{%
    \psline(-0.25,4)(-0.25,0.45)
    \psellipticarc(0,0.45)(0.27,0.15){180}{0}
    \psline(0.25,0.45)(0.25,4)(-0.25,4)}
  % Regulating flame
  \psframe[fillstyle=solid,fillcolor=lightgray](-0.5,1)(0.5,2)
  \psdots[dotstyle=o,dotscale=1.25](-0.3,1.5)(0,1.5)(0.3,1.5)
  % Small gaz tube
  \pscustom[fillstyle=solid,fillcolor=Orange]{%
    \psline(-2.5,0.6)(-0.25,0.6)
    \psellipticarc(-0.25,0.7)(0.1,0.12){-90}{90}
    \psline(-0.25,0.8)(-2.5,0.8)}
  % Flame
  \rput(0,4){%
    \psset{linestyle=none,fillstyle=gradient,gradmidpoint=0}%
    \psclip{\psbezier[gradbegin=LightOrange,gradend=yellow]%
                    (-0.25,0)(-0.35,0.5)(-0.4,0.75)
                    (-0.35,1)(-0.25,1.5)(0.5,2)
                    (0.25,1.5)(0.35,1)(0.4,0.75)
                    (0.35,0.5)(0.25,0)(0,0)}
      \pspolygon[gradbegin=blue,gradend=white](-0.25,0)(0.25,0)(0,1)
    \endpsclip}}}}

\newpsstyle{SulphuricAcid}{linestyle=none,fillstyle=solid,
  fillcolor=LemonChiffon}

\newpsstyle{PotassiumPermanganate}{linestyle=none,fillstyle=gradient,
  gradmidpoint=1,gradbegin=white,gradend=DarkViolet}

\newpsstyle{SulphuricDioxyde}{fillstyle=solid,fillcolor=Gold}

\psset{Substance=coloredbubbles,LiquidSurface=wavy}%
% Table
\psline[linewidth=0.1](-5,0)(5.7,0)
\psframe[linestyle=none,fillstyle=vlines](-5,0)(5.7,-1)
% Bunsen burner
\PstLaboBunsenBurner(-2.5,0)
% Tube on right
\PstLaboTestTube[Angle=30,
  LiquidType1=PotassiumPermanganate,LiquidLevel1=0.5,
  Substance=coloredbubbles,
  slopebegin=DarkViolet,slopeend=white](5.2,0.1)%
% Becher
\PstLaboTestTube[yunit=0.8,Form=becher](4.7,0.05)%
% Round bottom flask on left
\PstLaboTestTube[Form=roundbottom,Angle=-60,
  Stopper=true,Clip=true,
  EscapeTube=bent,EscapeTubeLength=8.7,doublecolor=PaleYellow,
  LiquidType1=SulphuricAcid,LiquidLevel1=0.7,
  Substance=bubbles,SubstanceNumber=30,
  SubstanceStyle=SulphuricDioxyde](-2.5,5.1)%
% Adjustment to redraw the external neck of the right tube
\rput{30}(5.45,0.14){%
  \psellipticarc[linewidth=0.08](0,3)(0.55,0.1){180}{0}}
% Adjustment to redraw the external neck of the becher
\rput(4.7,2.055){%
  \psellipticarc[yunit=0.8,linewidth=0.08](1.1,0.1){180}{0}}
% Labels
\psset{linewidth=0.05,arrows=->,arrowscale=2}%
\psline(-0.8,1.5)(-3,5.8)
  \uput[-90](-0.8,1.5){%
    \shortstack{Sulfuric acid\\$\mathrm{H_2SO_4}$}}
\psline(1,1.5)(-2.2,6.2)
  \uput[-90](1,1.5){\shortstack{Sulfur\\$\mathrm{SO}$}}
\psline(3.3,6)(3.3,3.8)
  \uput[90](3.3,6){\shortstack{Sulfur dioxyde\\$\mathrm{SO_2}$}}
\psline(5.4,7)(5.4,0.6)
  \uput[90](5.4,7){%
    \shortstack{Potassium permanganate\\$\mathrm{KMnO_4}$}}
```



64.3 Advanced example: Gantt charts

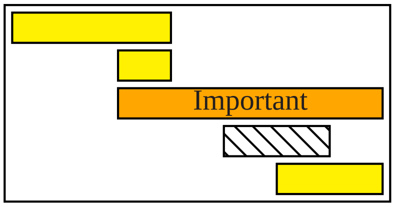
This is another complex example which builds high-level tools to draw simple Gantt charts. We will define a macro (\PstGanttTask) to describe tasks and will show how to define an environment (PstGanttChart) to manage all these defined tasks. Both must of course accept optional parameters.

```
% Parameters definition
% ================

% Intervals to show?
\newif\ifPstGantt@ChartShowIntervals
\define@key{psset}{ChartShowIntervals}[true]{%
\@nameuse{PstGantt@ChartShowIntervals#1}}

% Style for the tasks
\define@key{psset}{TaskStyle}{\edef\PstGantt@TaskStyle{#1}}

% Name for unit interval
\define@key{psset}{ChartUnitIntervalName}{%
\edef\PstGantt@ChartUnitIntervalName{#1}}

% Name for basic unit
\define@key{psset}{ChartUnitBasicIntervalName}{%
\edef\PstGantt@ChartUnitBasicIntervalName{#1}}

% Unit interval for the tasks
% (7 for a week, 30 for a month, etc.)
% Warning: define it before “TaskUnitType”!
\define@key{psset}{TaskUnitIntervalValue}{%
\pst@getint{#1}{\PstGantt@TaskUnitIntervalValue}}

% Unit type for the tasks
% (“UnitIntervalName” or “UnitBasicIntervalName”)
\define@key{psset}{TaskUnitType}{%
\edef\PstGantt@TaskUnitValue{#1}%
% Validation of the parameter
\ifx\PstGantt@TaskUnitValue\PstGantt@ChartUnitIntervalName
  \edef\PstGantt@TaskUnitValue{%
    \PstGantt@TaskUnitIntervalValue}%
\else
  \ifx\PstGantt@TaskUnitValue%
          \PstGantt@ChartUnitBasicIntervalName
    \def\PstGantt@TaskUnitValue{1}%
  \else
    {\@pstrickserr{GanttTaskUnitType must be
       ‘\PstGantt@ChartUnitIntervalName’
```

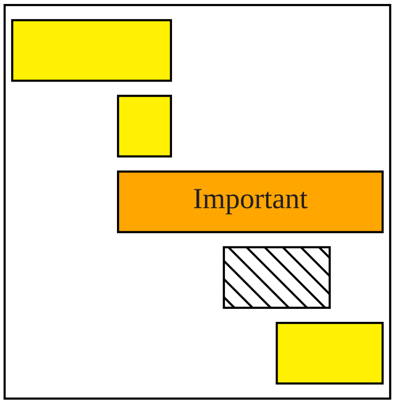
```
          or ‘\PstGantt@ChartUnitBasicIntervalName’
          (and not ‘\PstGantt@TaskUnitValue’)}\@eha}%
  \fi
\fi}

% Outside label for the tasks
\define@key{psset}{TaskOutsideLabel}{%
\def\PstGantt@TaskOutsideLabel{#1}}

% Inside label for the tasks
\define@key{psset}{TaskInsideLabel}{%
\def\PstGantt@TaskInsideLabel{#1}}

% Maximum outside size label for the tasks
% (in unit “TaskUnitType” !)
\define@key{psset}{TaskOutsideLabelMaxSize}{%
\pst@getint{#1}{\PstGantt@TaskOutsideLabelMaxSize}}

% Default values
% ==========
% Don’t show intervals, default task style, unit for tasks
% is a week (so 7 days), no outside and inside labels
\setkeys{psset}{%
ChartShowIntervals=false,TaskStyle=TaskStyleDefault,
ChartUnitIntervalName=Week,ChartUnitBasicIntervalName=Day,
TaskUnitIntervalValue=7,TaskUnitType=Week,TaskOutsideLabel=,
TaskInsideLabel=,TaskOutsideLabelMaxSize=0}

% Task default style is yellow background
\newpsstyle{TaskStyleDefault}{fillstyle=solid,fillcolor=yellow}

% The environment PstGanttChart
% =======================

% Syntax:\PstGanttChart[parameters]{Nb of tasks}{Nb of days}
%          \endPstGanttChart
% or
% \begin{PstGanttChart}[parameters]{Nb of tasks}{Nb of days}
% \end{PstGanttChart}
\def\PstGanttChart{\def\pst@par{}\pst@object{PstGanttChart}}

\def\PstGanttChart@i#1#2{%
\bgroup
  \setkeys{psset}{unit=0.1}%
  \use@par% Assignment of local parameters
  %
```

```
% "pspicture" environment
\pst@cnta=\PstGantt@TaskOutsideLabelMaxSize
\multiply\pst@cnta\PstGantt@TaskUnitValue
%
\pst@cntb=#1
\multiply\pst@cntb by 5
\advance\pst@cntb\@ne
%
\pst@cntc=#2
\multiply\pst@cntc\PstGantt@TaskUnitValue
\advance\pst@cntc\tw@
%
\ifPstGantt@ChartShowIntervals
  \pspicture(-\pst@cnta,-\pst@cntb)(\pst@cntc,\@two)
\else
  \pspicture(-\pst@cnta,-\pst@cntb)(\pst@cntc,\z@)
\fi
\psframe(\z@,-\pst@cntb)(\pst@cntc,\z@)
%
\ifPstGantt@ChartShowIntervals
  % We will show the intervals
  \pst@cnta=#2
  \multiply\pst@cnta\PstGantt@TaskUnitValue
  \divide\pst@cnta\PstGantt@TaskUnitIntervalValue
  \advance\pst@cnta\@ne
  %
  \pst@cntb=#1
  \multiply\pst@cntb by 5
  \advance\pst@cntb\@ne
  %
  \pst@dima=\PstGantt@TaskUnitIntervalValue\p@
  \divide\pst@dima\tw@
  \advance\pst@dima\@ne\p@
  %
  \multido{\iInterval=\@ne+\@ne,
         \iIntervalPos=\@ne+
          \PstGantt@TaskUnitIntervalValue,
         \rIntervalPos=\pst@number{\pst@dima}+%
           \PstGantt@TaskUnitIntervalValue}{\pst@cnta}{%
    \ifnum\iInterval=\pst@cnta
      \psline(\iIntervalPos,\z@)(\iIntervalPos,1.5)
      \psline[linestyle=dotted](\iIntervalPos,-\pst@cntb)
                         (\iIntervalPos,\z@)
    \else
      \rput(\rIntervalPos,\@ne){%
        \PstGantt@ChartUnitIntervalName{} \iInterval}
        \psline(\iIntervalPos,\z@)(\iIntervalPos,1.5)
        \psline[linestyle=dotted](\iIntervalPos,-\pst@cntb)
                                   (\iIntervalPos,\z@)
      \fi}
  \fi}

\def\endPstGanttChart{%
  \endpspicture       % End of "pspicture" environment
\egroup}

% The macro \PstGanttTask
% ===================

\newcount\PstGantt@TaskCnt
\PstGantt@TaskCnt=\z@

% Syntax: \PstGanttTask[parameters]{Start}{Length}
\def\PstGanttTask{\def\pst@par{}\pst@object{PstGanttTask}}

\def\PstGanttTask@i#1#2{%
\advance\PstGantt@TaskCnt\m@ne % To increment globally
\bgroup
  \use@par% Assignment of local parameters
  % Frame
  \pst@cnta=\PstGantt@TaskUnitValue
  \multiply\pst@cnta by #1
  \advance\pst@cnta\@ne
  %
  \pst@cntb=\PstGantt@TaskUnitValue
  \multiply\pst@cntb by #2
  \advance\pst@cntb\pst@cnta
  %
  \pst@cntc=\PstGantt@TaskCnt
  \multiply\pst@cntc by 5
  %
  \pst@cntd=\pst@cntc
  \advance\pst@cntd by 4
  %
  \psframe[style=\PstGantt@TaskStyle](\pst@cnta,\pst@cntc)
                                        (\pst@cntb,\pst@cntd)
  % Inside label
  \ifx\PstGantt@TaskInsideLabel\@empty
  \else
    \pst@dima=\pst@cnta\p@
    \advance\pst@dima\pst@cntb\p@
    \divide\pst@dima\tw@
    %
    \pst@dimb=\pst@cntc\p@
    \advance\pst@dimb\pst@cntd\p@
    \divide\pst@dimb\tw@
    %
  \rput(\pst@number{\pst@dima},\pst@number{\pst@dimb}){%
    \PstGantt@TaskInsideLabel}
  \fi
  % Outside label
  \ifx\PstGantt@TaskOutsideLabel\@empty
  \else
    \pst@dima=\pst@cntc\p@
    \advance\pst@dima\pst@cntd\p@
    \divide\pst@dima\tw@
    \rput[r](-1.5,\pst@number{\pst@dima}){%
      \PstGantt@TaskOutsideLabel}
  \fi
\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 LaTeX flavor syntax, which of course must be adapted if PLAIN TeX or ConTeXt is used.)

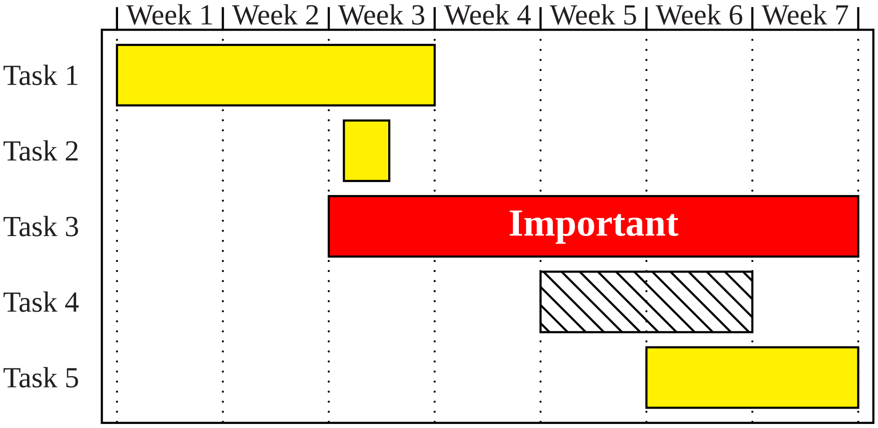


```
\newpsstyle{Important}{fillstyle=solid,fillcolor=Orange}
\newpsstyle{NotImportant}{fillstyle=vlines}

\begin{PstGanttChart}{5}{7}
  \PstGanttTask{0}{3}
  \PstGanttTask{2}{1}
  \PstGanttTask[TaskStyle=Important,
                TaskInsideLabel=Important]{2}{5}
  \PstGanttTask[TaskStyle=NotImportant]{4}{2}
  \PstGanttTask{5}{2}
\end{PstGanttChart}
```

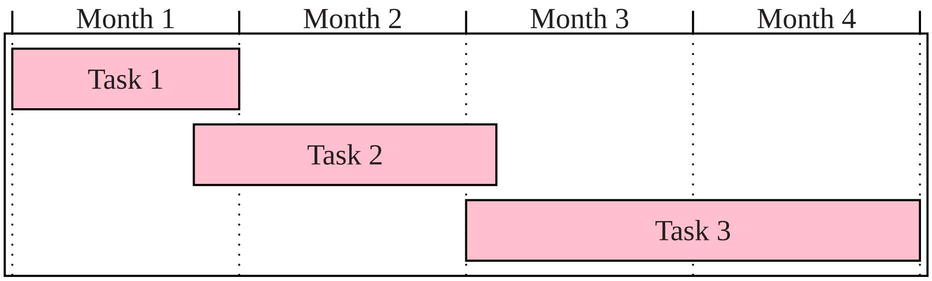


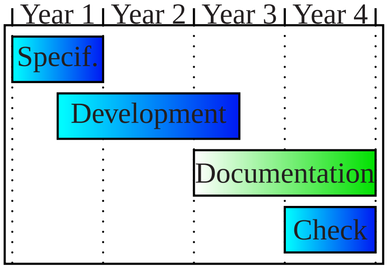
```
\begin{PstGanttChart}[yunit=2]{5}{7}% Same with double "yunit"
  \PstGanttTask{0}{3}
  \PstGanttTask{2}{1}
  \PstGanttTask[TaskStyle=Important,
                TaskInsideLabel=Important]{2}{5}
  \PstGanttTask[TaskStyle=NotImportant]{4}{2}
  \PstGanttTask{5}{2}
\end{PstGanttChart}
```

```
\newpsstyle{Important}{fillstyle=solid,fillcolor=red}

\begin{PstGanttChart}[unit=2,TaskOutsideLabelMaxSize=1,
                                  ChartShowIntervals=true]{5}{7}
   \PstGanttTask[TaskOutsideLabel={Task 1}]{0}{3}
   \PstGanttTask[TaskOutsideLabel={Task 2},TaskUnitType=Day]
                        {15}{3} % 3 days starting at day 15
   \PstGanttTask[TaskStyle=Important,TaskOutsideLabel={Task 3},
       TaskInsideLabel={\Large\textcolor{white}{\textbf{Important}}}]{2}{5}
   \PstGanttTask[TaskStyle=NotImportant,
                          TaskOutsideLabel={Task 4}]{4}{2}
   \PstGanttTask[TaskOutsideLabel={Task 5}]{5}{2}
\end{PstGanttChart}
```



```
\newpsstyle{TaskStyle}{fillstyle=solid,fillcolor=Pink}

\begin{PstGanttChart}[yunit=2,ChartUnitIntervalName=Month,
                                  TaskUnitIntervalValue=30,TaskUnitType=Month,
                                  ChartShowIntervals,TaskStyle=TaskStyle]{3}{4}
   \PstGanttTask[TaskInsideLabel={Task 1}]{0}{1}
   \PstGanttTask[TaskInsideLabel={Task 2},TaskUnitType=Day]
                        {24}{40} % 40 days starting at day 24
   \PstGanttTask[TaskInsideLabel={Task 3}]{2}{2}
\end{PstGanttChart}
```





```
\newpsstyle{MyTaskStyleA}{fillstyle=gradient,gradmidpoint=0,
  gradangle=90}
\newpsstyle{MyTaskStyleB}{fillstyle=gradient,gradmidpoint=0,
  gradangle=90,gradbegin=ForestGreen,gradend=white}

\begin{PstGanttChart}[yunit=1.5,ChartUnitIntervalName=Year,
    ChartShowIntervals=true,ChartUnitBasicIntervalName=Month,
    TaskUnitIntervalValue=12,TaskUnitType=Year,
    TaskStyle=MyTaskStyleA]{4}{4}
  \PstGanttTask[TaskInsideLabel={Specif.}]{0}{1}
  \PstGanttTask[TaskInsideLabel={Development},
    TaskUnitType=Month]{6}{24} % 24 months since month 6
  \PstGanttTask[TaskStyle=MyTaskStyleB,
                TaskInsideLabel={Documentation}]{2}{2}
  \PstGanttTask[TaskInsideLabel={Check}]{3}{1}
\end{PstGanttChart}
```

64.4 Advanced example: overlapped colored surfaces[53]

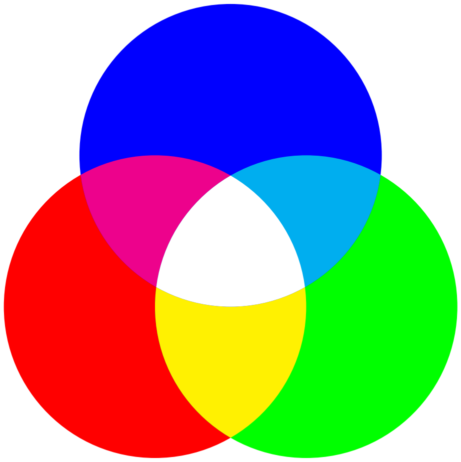


Figure 3: Basic drawing of overlapped circles with colors in additive synthesis

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 ressources (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.

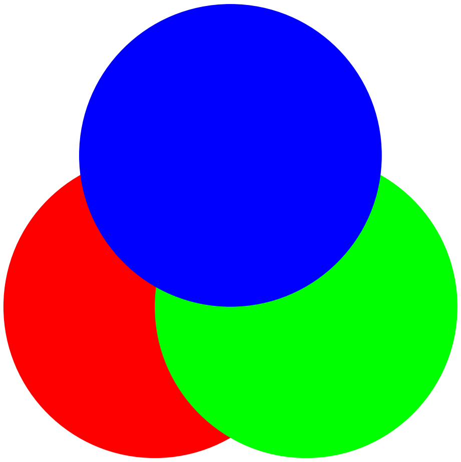
---

[53] This 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.



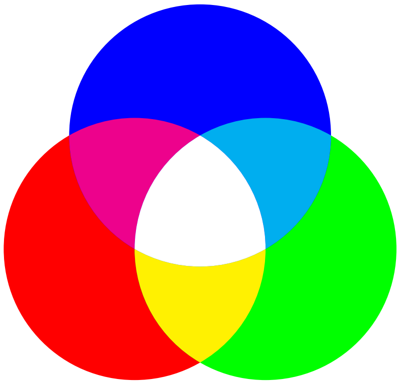
```
\psset{linestyle=none,fillstyle=solid}
% Drawing of the circles
\pscircle[fillcolor=red](0,0){2}
\pscircle[fillcolor=green](2,0){2}
\pscircle[fillcolor=blue](1,2){2}
```

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):

```
\SpecialCoor
\psset{linestyle=none,fillstyle=solid}
% sqrt(1 + 2) = 1.732
\pscircle[fillcolor=red](1;210){1.732}%     Red circle
\pscircle[fillcolor=green](1;330){1.732}%  Green circle
\pscircle[fillcolor=blue](1;90){1.732}%     Blue circle
% Yellow intersection of Red and Green circles (R + G = Y)
\pscustom[fillcolor=yellow]{%
  \psarc(1;-30){1.732}{180}{240}     % B
  \psarc(1;-150){1.732}{-60}{0}       % A
  \psarcn(1;90){1.732}{300}{240}}    % C
% Cyan intersection of Green and Blue circles (G + B = C)
\pscustom[fillcolor=cyan]{%
  \psarc(1;90){1.732}{-60}{0}          % C
  \psarc(1;-30){1.732}{60}{120}       % B
  \psarcn(1;-150){1.732}{60}{0}}     % A
% Magenta intersection of Red and Blue circles (R + B = M)
\pscustom[fillcolor=magenta]{%
  \psarc(1;90){1.732}{180}{240}       % C
  \psarcn(1;-30){1.732}{180}{120}   % B
  \psarc(1;-150){1.732}{60}{120}}   % A
% White intersection of Red, Green and Blue circles (R+G+B=W)
\pscustom[fillcolor=white]{%
  \psarc(1;90){1.732}{-120}{-60}     % C
  \psarc(1;-150){1.732}{0}{60}         % A
  \psarc(1;-30){1.732}{120}{180}}   % B
```

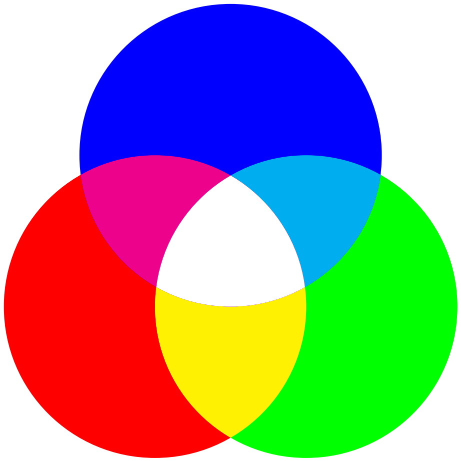


We can see that this was rather easy to do, but also rather painful, because we must compute all the positions of the points delimiting the various arcs, and we are limited to using *predefined* resulting colors (that is to say that, concerning the colors, we must know the solution before to start to program!) The kind of surfaces is also completey 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:

```
\psset{linestyle=none}
\def\PstCircleA{\pscircle(0,0){2}}
\def\PstCircleB{\pscircle(2,0){2}}
\def\PstCircleC{\pscircle(1,2){2}}

{\psset{fillstyle=solid,fillcolor=red}\PstCircleA}%     Red circle
{\psset{fillstyle=solid,fillcolor=green}\PstCircleB}%  Green circle
{\psset{fillstyle=solid,fillcolor=blue}\PstCircleC}%    Blue circle
%
% Yellow intersection of Red and Green circles (R + G = Y)
```

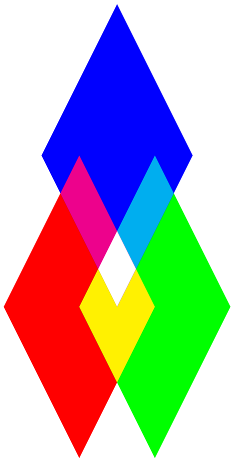
```
\psclip{\PstCircleB}
  \psset{fillstyle=solid,fillcolor=yellow}\PstCircleA
\endpsclip
% Cyan intersection of Green and Blue circles (G + B = C)
\psclip{\PstCircleB}
  \psset{fillstyle=solid,fillcolor=cyan}\PstCircleC
\endpsclip
% Magenta intersection of Red and Blue circles (R + B = M)
\psclip{\PstCircleC}
  \psset{fillstyle=solid,fillcolor=magenta}\PstCircleA
\endpsclip
% White intersection of Red, Green and Blue circles (R+G+B=W)
\psclip{\PstCircleB\PstCircleC}
  \psset{fillstyle=solid,fillcolor=white}\PstCircleA
\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.

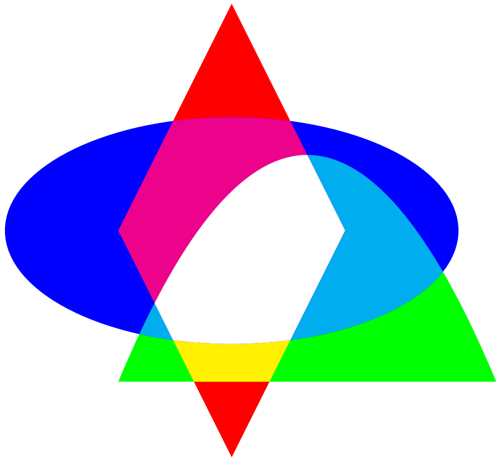
```
% The three surfaces must be defined as \PstSurfaceA,
% \PstSurfaceB and \PstSurfaceC

\def\PstColorSynthesis{{%
\psset{linestyle=none}%
{\psset{fillstyle=solid,fillcolor=red}\PstSurfaceA}%     Red circle
{\psset{fillstyle=solid,fillcolor=green}\PstSurfaceB}%  Green circle
{\psset{fillstyle=solid,fillcolor=blue}\PstSurfaceC}%    Blue circle
%
% Yellow intersection of Red and Green surfaces (R + G = Y)
\psclip{\PstSurfaceB}
  \psset{fillstyle=solid,fillcolor=yellow}\PstSurfaceA
\endpsclip
% Cyan intersection of Green and Blue surfaces (G + B = C)
\psclip{\PstSurfaceB}
   \psset{fillstyle=solid,fillcolor=cyan}\PstSurfaceC
\endpsclip
% Magenta intersection of Red and Blue surfaces (R + B = M)
\psclip{\PstSurfaceC}
   \psset{fillstyle=solid,fillcolor=magenta}\PstSurfaceA
\endpsclip
% White intersection of Red, Green and Blue surfaces
% (R + G + B = W)
\psclip{\PstSurfaceB\PstSurfaceC}
   \psset{fillstyle=solid,fillcolor=white}\PstSurfaceA
\endpsclip}}

\def\PstSurfaceA{\psdiamond(0,0)(1,2)}
\def\PstSurfaceB{\psdiamond(1,0)(1,2)}
\def\PstSurfaceC{\psdiamond(0.5,2)(1,2)}

\PstColorSynthesis
```



Now we can transform it in a PSTricks object, as usual.

```
% Surface parameters
\define@key{psset}{SurfaceA}{\def\PstColorSynthesis@SurfaceA{#1}}
\define@key{psset}{SurfaceB}{\def\PstColorSynthesis@SurfaceB{#1}}
\define@key{psset}{SurfaceC}{\def\PstColorSynthesis@SurfaceC{#1}}

\def\PstColorSynthesis{\def\pst@par{}\pst@object{PstColorSynthesis}}

\def\PstColorSynthesis@i{{%
\setkeys{psset}{linestyle=none}%
\use@par% Assignment of local parameters
\bgroup
   \setkeys{psset}{fillstyle=solid,fillcolor=red}%
   \PstColorSynthesis@SurfaceA% Red surface
   \setkeys{psset}{fillcolor=green}%
   \PstColorSynthesis@SurfaceB% Green surface
```

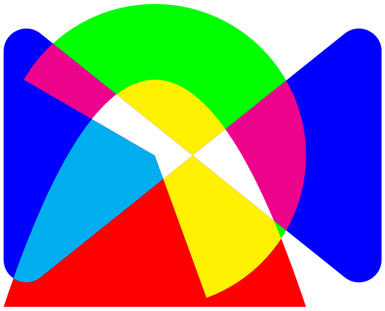
```
  \setkeys{psset}{fillcolor=blue}%
  \PstColorSynthesis@SurfaceC% Blue surface
\egroup
%
% Yellow intersection of Red and Green surfaces (R + G = Y)
\psclip{\PstColorSynthesis@SurfaceB}
  \setkeys{psset}{fillstyle=solid,fillcolor=yellow}%
  \PstColorSynthesis@SurfaceA
\endpsclip
% Cyan intersection of Green and Blue surfaces (G + B = C)
\psclip{\PstColorSynthesis@SurfaceB}
  \setkeys{psset}{fillstyle=solid,fillcolor=cyan}%
  \PstColorSynthesis@SurfaceC
\endpsclip
% Magenta intersection of Red and Blue surfaces (R + B = M)
\psclip{\PstColorSynthesis@SurfaceC}
  \setkeys{psset}{fillstyle=solid,fillcolor=magenta}%
  \PstColorSynthesis@SurfaceA
\endpsclip
% White intersection of Red, Green and Blue surfaces
% (R + G + B = W)
\psclip{\PstColorSynthesis@SurfaceB\PstColorSynthesis@SurfaceC}
  \setkeys{psset}{fillstyle=solid,fillcolor=white}%
  \PstColorSynthesis@SurfaceA
\endpsclip}}

\PstColorSynthesis[SurfaceA={\psdiamond(1.5,3)},
  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.

```
% The seven color names to use
\define@key{psset}{ColorA}{\def\PstColorSynthesis@ColorA{#1}}
```

```
\define@key{psset}{ColorB}{\def\PstColorSynthesis@ColorB{#1}}
\define@key{psset}{ColorC}{\def\PstColorSynthesis@ColorC{#1}}
\define@key{psset}{ColorD}{\def\PstColorSynthesis@ColorD{#1}}
\define@key{psset}{ColorE}{\def\PstColorSynthesis@ColorE{#1}}
\define@key{psset}{ColorF}{\def\PstColorSynthesis@ColorF{#1}}
\define@key{psset}{ColorG}{\def\PstColorSynthesis@ColorG{#1}}

\def\PstColorSynthesis@i{{%
\setkeys{psset}{linestyle=none}%
\use@par% Assignment of local parameters
% The initial surfaces in their original colors
\bgroup
  \setkeys{psset}{fillstyle=solid,fillcolor=\PstColorSynthesis@ColorA}%
  \PstColorSynthesis@SurfaceA
  \setkeys{psset}{fillcolor=\PstColorSynthesis@ColorB}%
  \PstColorSynthesis@SurfaceB
  \setkeys{psset}{fillcolor=\PstColorSynthesis@ColorC}%
  \PstColorSynthesis@SurfaceC
\egroup
% We redraw them in a combinatorial way, with clipping
\PstColorSynthesis@ClippedSurfaces{\PstColorSynthesis@SurfaceA}
  {\PstColorSynthesis@SurfaceB}{}{\PstColorSynthesis@ColorD}%
\PstColorSynthesis@ClippedSurfaces{\PstColorSynthesis@SurfaceA}
  {\PstColorSynthesis@SurfaceC}{}{\PstColorSynthesis@ColorE}%
\PstColorSynthesis@ClippedSurfaces{\PstColorSynthesis@SurfaceB}
  {\PstColorSynthesis@SurfaceC}{}{\PstColorSynthesis@ColorF}%
\PstColorSynthesis@ClippedSurfaces{\PstColorSynthesis@SurfaceA}
  {\PstColorSynthesis@SurfaceB}{\PstColorSynthesis@SurfaceC}
  {\PstColorSynthesis@ColorG}}}

\def\PstColorSynthesis@ClippedSurfaces#1#2#3#4{%
% We draw the first surface #1 in color #4, clipped by the one
% or two others #2 and #3
\psclip{#2#3}
  \setkeys{psset}{fillstyle=solid,fillcolor=#4}%
  #1
\endpsclip}

\PstColorSynthesis[SurfaceA={\parabola(-1,-1)(1,2)},
  SurfaceB={\pswedge(1,1){2}{-70}{150}},
  SurfaceC={\pspolygon[linearc=0.3](-1,-1)(-1,3)(4,-1)(4,3)},
  ColorA=red,ColorB=green,ColorC=blue,ColorD=yellow,ColorE=cyan,
  ColorF=magenta,ColorG=white]
```



So, we have already make huge progress! Moreover, the resulting code is far more general, powerful and in fact easier to write as it involve 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 technics.[54]

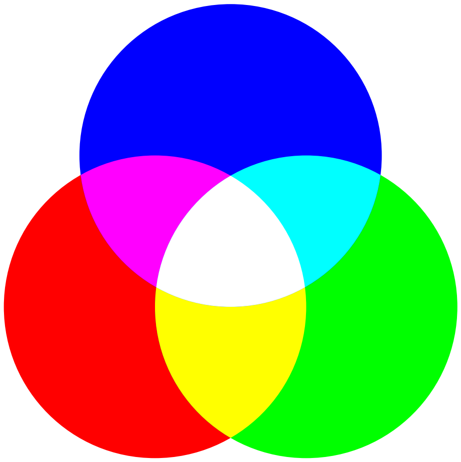
```
% Additive color synthesis (correct results only for sums
% of RGB components <= 1)
% Colors must be defined according the RGB model
% (so, for instance, to use “black”, it must be redefined in RGB)

\def\PstColorSynthesis@i{{%
\setkeys{psset}{linestyle=none}%
\use@par% Assignment of local parameters
\bgroup
  \setkeys{psset}{fillstyle=solid,fillcolor=\PstColorSynthesis@ColorA}%
  \PstColorSynthesis@SurfaceA
  \setkeys{psset}{fillcolor=\PstColorSynthesis@ColorB}%
  \PstColorSynthesis@SurfaceB
  \setkeys{psset}{fillcolor=\PstColorSynthesis@ColorC}%
  \PstColorSynthesis@SurfaceC
\egroup
% We redraw them in a combinatorial way, with clipping,
% and computing the resulting color of overlapped surfaces
% two by two then the three together
\PstColorSynthesis@ClippedSurfaces{A}{B}{NoSurface}%
\PstColorSynthesis@ClippedSurfaces{A}{C}{NoSurface}%
\PstColorSynthesis@ClippedSurfaces{B}{C}{NoSurface}%
\PstColorSynthesis@ClippedSurfaces{A}{B}{C}}}

% For the cases where there are only two surfaces
% in the combination
\def\PstColorSynthesis@ColorNoSurface{}

% We draw the first surface #1 clipped by the one
% or two others #2 and #3, computing it resulting color
```

---

[54] Note that the proposed solution here is based on the color package, which can be used with Plain TeX and LaTeX, but with ConTeXt, modifications must be made to implement color management.

```
\def\PstColorSynthesis@ClippedSurfaces#1#2#3{%
% We compute the “mixed” color, component by component
\def\PstColorSynthesis@MixedColorR{0}%
\def\PstColorSynthesis@MixedColorG{0}%
\def\PstColorSynthesis@MixedColorB{0}%
\PstColorSynthesis@MixedColor{%
   \csname PstColorSynthesis@Color#1\endcsname}%
\PstColorSynthesis@MixedColor{%
   \csname PstColorSynthesis@Color#2\endcsname}%
\PstColorSynthesis@MixedColor{%
   \csname PstColorSynthesis@Color#3\endcsname}%
% We draw the first surface, clipped by the other ones
\psclip{\csname PstColorSynthesis@Surface#2\endcsname%
         \csname PstColorSynthesis@Surface#3\endcsname}
   \definecolor{MixedColor}{rgb}{\PstColorSynthesis@MixedColorR,
      \PstColorSynthesis@MixedColorG,
      \PstColorSynthesis@MixedColorB}%
   \setkeys{psset}{fillstyle=solid,fillcolor=MixedColor}%
   \csname PstColorSynthesis@Surface#1\endcsname
\endpsclip}

\def\PstColorSynthesis@MixedColor#1{%
\edef\@tempa{#1}%
\ifx\@tempa\@empty
\else
   % We ask for the values of the 3 components of the color.
   % This is the difficult line:
   %    - in the “color” package, a color is defined by the macro
   %       \color@ColorName,
   %    - so, we must build the macro name \color@#1 and call
   %       the macro \PstColorSynthesis@MixedColor@i with the
   %       result of the execution of it.
   %
   % Note also that, if we want to manage too the “cmyk” color
   % model, this macro and the next one must me modified, as
   % \color@ColorName will return one argument more:
   % \pst@expandafter\PstColorSynthesis@MixedColor@i{%
   %    \csname\string\color @#1\endcsname} \@nil
   % ...
   % \def\PstColorSynthesis@MixedColor@i#1 #2 #3 #4 #5\@nil{%
   %
   \pst@expandafter\PstColorSynthesis@MixedColor@i{%
      \csname\string\color@#1\endcsname}\@nil
\fi}

\def\PstColorSynthesis@MixedColor@i#1 #2 #3 #4\@nil{%
% We receive the name of the color model (“rgb” is expected
% here) and the three color components.
% We add the values to the respective components of the new
% color to compute.
\PstColorSynthesis@MixedColor@ii{%
   \PstColorSynthesis@MixedColorR}{#2}%
\PstColorSynthesis@MixedColor@ii{%
```

```
   \PstColorSynthesis@MixedColorG}{#3}%
\PstColorSynthesis@MixedColor@ii{%
   \PstColorSynthesis@MixedColorB}{#4}}

\def\PstColorSynthesis@MixedColor@ii#1#2{%
% As these values are real numbers, we use a dimension
% register, then we assign the computed value in a macro,
% converting it from a dimension to a number
\pst@dimg=#1\p@
\advance\pst@dimg by #2\p@
\pst@dimtonum{\pst@dimg}{#1}}

\PstColorSynthesis[SurfaceA={\pscircle{2}},
   SurfaceB={\pscircle(2,0){2}},SurfaceC={\pscircle(1,2){2}},
   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]

```
% We draw the first surface #1 clipped by the one
% or two others #2 and #3, computing it resulting color
\def\PstColorSynthesis@ClippedSurfaces#1#2#3{%
% We compute the “mixed” color, component by component
```

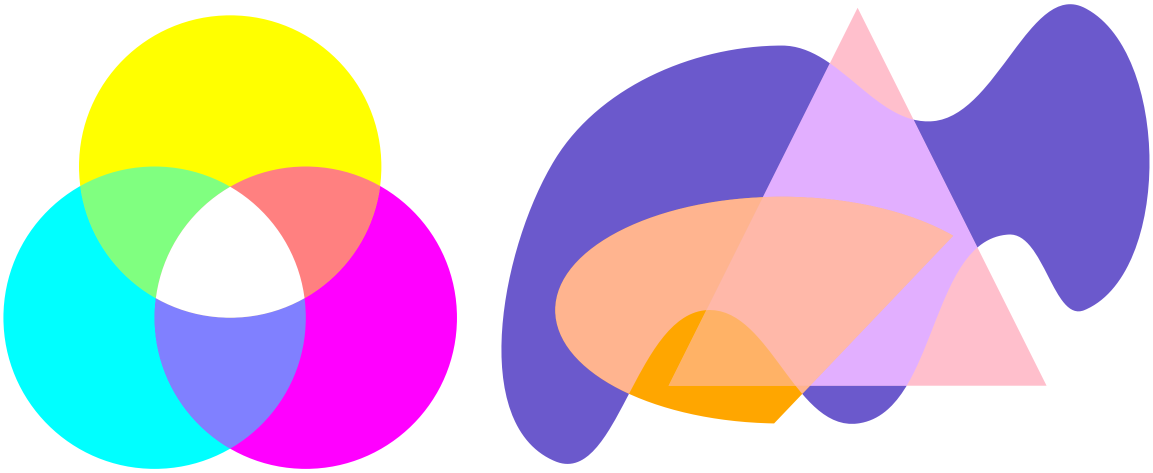
---

[55] For 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, substractive, 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.

```
\def\PstColorSynthesis@MixedColorR{0}%
\def\PstColorSynthesis@MixedColorG{0}%
\def\PstColorSynthesis@MixedColorB{0}%
% \pst@dimd will contain the maximum value of the three
% components.
\pst@dimd=\z@
\PstColorSynthesis@MixedColor{%
  \csname PstColorSynthesis@Color#1\endcsname}%
\PstColorSynthesis@MixedColor{%
  \csname PstColorSynthesis@Color#2\endcsname}%
\PstColorSynthesis@MixedColor{%
  \csname PstColorSynthesis@Color#3\endcsname}%
% We must test if the maximum of the new computed
% component is greater than 1, in which case we divide all
% the three components by this value for additive synthesis.
\ifdim\pst@dimd>\@ne\p@
  \PstColorSynthesis@MixedColor@iii{%
    \PstColorSynthesis@MixedColorR}%
  \PstColorSynthesis@MixedColor@iii{%
    \PstColorSynthesis@MixedColorG}%
  \PstColorSynthesis@MixedColor@iii{%
    \PstColorSynthesis@MixedColorB}%
\fi
% We draw the first surface, clipped by the other ones
\psclip{\csname PstColorSynthesis@Surface#2\endcsname%
        \csname PstColorSynthesis@Surface#3\endcsname}
  \definecolor{MixedColor}{rgb}{\PstColorSynthesis@MixedColorR,
    \PstColorSynthesis@MixedColorG,
    \PstColorSynthesis@MixedColorB}%
  \setkeys{psset}{fillstyle=solid,fillcolor=MixedColor}%
  \csname PstColorSynthesis@Surface#1\endcsname
\endpsclip}

\def\PstColorSynthesis@MixedColor@i#1 #2 #3 #4\@nil{%
% We receive the name of the color model (“rgb” is expected
% here and the three color components.)
% We add the values to the respective components of the new
% color to compute.
\PstColorSynthesis@MixedColor@ii{%
  \PstColorSynthesis@MixedColorR}{#2}%
\PstColorSynthesis@MixedColor@ii{%
  \PstColorSynthesis@MixedColorG}{#3}%
\PstColorSynthesis@MixedColor@ii{%
  \PstColorSynthesis@MixedColorB}{#4}}

\def\PstColorSynthesis@MixedColor@ii#1#2{%
% As these values are real numbers, we use dimension
% registers, then we assign the computed value in a macro,
% converting it from a dimension to a number.
% We also keep in \pst@dimd the maximum of the values.
\pst@dimg=#1\p@
\advance\pst@dimg by #2\p@
\pst@dimtonum{\pst@dimg}{#1}%
```

```
\ifdim\pst@dimg>\pst@dimd
    \pst@dimd=\pst@dimg
\fi}

\def\PstColorSynthesis@MixedColor@iii#1{%
\pst@divide{#1\p@}{\pst@dimd}{#1}%
% We must take care of possible rounding problems with
% \pst@divide (for instance, 1.8/1.8 give 1.0001)
\ifdim#1\p@>\@ne\p@
    \def#1{1}%
\fi}

% We redefine Cyan, Magenta and Yellow in the “rgb” model
\definecolor{Cyan}{rgb}{0,1,1}%
\definecolor{Magenta}{rgb}{1,0,1}%
\definecolor{Yellow}{rgb}{1,1,0}%
\PstColorSynthesis[SurfaceA={\pscircle{2}},
    SurfaceB={\pscircle(2,0){2}},SurfaceC={\pscircle(1,2){2}},
    ColorA=Cyan,ColorB=Magenta,ColorC=Yellow]
\PstColorSynthesis[SurfaceA={\psccurve(-3,1)(0,2.5)(2,1.5)(4,3)(4,-1)
                                      (3,0)(1,-2.5)(-1,-1)(-3,-3)},
    SurfaceB={\psellipticarc(0,-1)(3,1.5){41}{-92}},
    SurfaceC={\pstriangle(1,-2)(5,5)},
    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.)

```
%% ‘pst-li3d.ins’
%%
%% Docstrip installation instruction file for docstyle ‘pst-li3d’
%%
%% Denis Girou (CNRS/IDRIS - France) <Denis.Girou@idris.fr>
%% and Peter Kleiweg (Rijksuniversiteit Groningen - Nederlands)
%% <kleiweg@let.rug.nl>
%%
%% July 10, 2003

\def\batchfile{pst-li3d.ins}
```

---

[56] We 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

```
\input docstrip.tex
\keepsilent
\Msg{*** Generating the ‘pst-li3d’ package ***}
\askforoverwritefalse
\generate{\file{pst-li3d.tex}{\from{pst-li3d.dtx}{pst-li3d}}}
\generate{\file{pst-li3d.sty}{\from{pst-li3d.dtx}{latex-wrapper}}}

% We must not write the “\endinput” line at the end
% of this external file!
\def\EpsFilePostamble{\MetaPrefix\space End of file ‘\outFileName’.}
\usepostamble\EpsFilePostamble
\generate{\file{pst-li3d.pro}{\from{pst-li3d.dtx}{postscript-header}}}

\ifToplevel{%
\Msg{********************************************************}
\Msg{*}
\Msg{* To finish the installation you have to move the files}
\Msg{* pst-li3d.sty and pst-li3d.tex in a directory/folder searched}
\Msg{* by TeX and the file pst-li3d.pro in a directory/folder}
\Msg{* searched by the DVI converter or the compiler, depending}
\Msg{* of your installation.}
\Msg{*}
\Msg{* To produce the documentation, run the file ‘pst-li3d.dtx’}
\Msg{* through LaTeX.}
\Msg{*}
\Msg{* If you require the commented code, desactivating the}
\Msg{* OnlyDescription macro, you must recompile, execute:}
\Msg{* ‘makeindex -s gind.ist pst-li3d’}
\Msg{* ‘makeindex -s gglo.ist -o pst-li3d.gls pst-li3d.glo’}
\Msg{* and recompile.}
\Msg{*}
\Msg{********************************************************}
}

\endinput
%%
%% 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 `^^A`.

```
% \iffalse meta-comment, etc.
%%
%% Package ‘pst-li3d.dtx’
%%
%% Denis Girou (CNRS/IDRIS - France) <Denis.Girou@idris.fr>
%% and Peter Kleiweg (Rijksuniversiteit Groningen - Nederlands)
%% <kleiweg@let.rug.nl>
%%
%% July 10, 2003
%%
%% This program can be redistributed and/or modified under
%% the terms of the LaTeX Project Public License Distributed
%% from CTAN archives in directory macros/latex/base/lppl.txt.
%%
%% DESCRIPTION:
%%    ‘pst-li3d’ is a PSTricks package for three dimensional
%%    lighten effect on characters and PSTricks graphics.
%%
% \fi
```

The \changes macro allow to describe the various releases of the package.

```
% \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 LaTeX, and is to put as the argument of the macro.

```
% \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.

```
% \DoNotIndex{\@}
% \DoNotIndex{\begin,\begin@ClosedObj,\box}
% ...
% \DoNotIndex{\use@par,\use@pscode,\usepackage}
%
% ^^A For the index and changes log (if source code printed)
% \setcounter{IndexColumns}{2}
% \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.

---

[57] The 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{}
```

```
%
% ^^A .................... End of the documentation part ....................
```

4 Driver file

Then we include the driver file which will be used to generate the documentation.

```
% \section{Driver file}
%
%    The next bit of code contains the documentation driver file
% for \TeX{}, i.e., the file that will produce the documentation
% you are currently reading. It will be extracted from this file
% by the \texttt{docstrip} program.
%
%       \begin{macrocode}
%<*driver>
\documentclass{ltxdoc}
\GetFileInfo{pst-li3d.dtx}
\usepackage{fancyvrb}
\usepackage{multido}
\usepackage{pstcol}
\usepackage{pst-grad}
\usepackage{pst-li3d}
\usepackage{pst-plot}
\usepackage{pst-tree}
\usepackage{url}
\definecolor{DarkGreen}{cmyk}{1,0,1,0.8}
\definecolor{Gold}          {rgb}{1,0.84,0}
\definecolor{Violet}        {cmyk}{0.79,0.88,0,0}
\EnableCrossrefs
\CodelineIndex
\RecordChanges
\OnlyDescription                 % Comment it for implementation details
\hbadness=7000                   % Over and under full box warnings
\hfuzz=3pt
\begin{document}
   \DocInput{pst-li3d.dtx}
\end{document}
%</driver>
%       \end{macrocode}
```

5 ‘pst-li3d’ LaTeX wrapper

Then we define the short LaTeX wrapper file, which will be useful for LaTeX users.

```
% \section{\PstLightThreeDPackage{} \LaTeX{} wrapper}
%
%       \begin{macrocode}
```

```
%<*latex-wrapper>
\ProvidesPackage{pst-li3d}[2003/07/10  package  wrapper
                                 for  PSTricks  pst-li3d.tex]
\input{pst-li3d}
%</latex-wrapper>
%       \end{macrocode}
```

And now the generic TEX (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

```
%  \section{\PstLightThreeDPackage{}  code}
%
%       \begin{macrocode}
%<*pst-li3d>
%       \end{macrocode}
```

6.1 Preambule

Who we are.

```
%  \subsection{Preambule}
%
%     Who  we  are.
%
%       \begin{macrocode}
\def\FileVersion{1.0}
\def\FileDate{2003/07/10}
\message{`Pst-Light3d'  v\FileVersion,  \FileDate\space
              (Denis  Girou  and  Peter  Kleiweg)}
\csname  PSTLightThreeDLoaded\endcsname
\let\PSTLightThreeDLoaded\endinput
%       \end{macrocode}
```

Require the PSTricks package.

```
%     Require  the  PSTricks  package.
%
%       \begin{macrocode}
\ifx\PSTricksLoaded\endinput\else\input{pstricks}\fi
%       \end{macrocode}
```

David CARLISLE interface to the 'keyval' package.

```
%     David  \textsc{Carlisle}  interface  to  the  `\textsf{keyval}'
%  package.
%
```

```
%    \begin{macrocode}
\input{pst-key}
%    \end{macrocode}
```

Catcodes changes.

```
%   Catcodes changes.
%
%    \begin{macrocode}
\edef\PstAtCode{\the\catcode‘\@}
\catcode‘\@=11\relax
%    \end{macrocode}
```

We load the PostScript header file.

```
%   We load the PostScript header file.
%
%    \begin{macrocode}
\pstheader{pst-li3d.pro}
%    \end{macrocode}
```

6.2 Definition of the parameters

`LightThreeDXLength` will be the horizontal length of the frame added. It is a *length* value.

```
% \subsection{Definition of the parameters}
%
%   \texttt{LightThreeDXLength} will be the horizontal length of the
% frame added. It is a \emph{length} value.
%
%    \begin{macrocode}
\define@key{psset}{LightThreeDXLength}{%
\pst@getlength{#1}{\PstLightThreeD@XLength}}
%    \end{macrocode}
```

`LightThreeDYLength` will be the vertical length of the frame added. It is a *length* value.

```
%   \texttt{LightThreeDYLength} will be the vertical length of the
% frame added. It is a \emph{length} value.
%
%    \begin{macrocode}
\define@key{psset}{LightThreeDYLength}{%
\pst@getlength{#1}{\PstLightThreeD@YLength}}
%    \end{macrocode}
```

`LightThreeDLength` will define both the horizontal and the vertical length of the frame added. It is a *length* value.

```
%    \texttt{LightThreeDLength} will define both the horizontal and
% the vertical length of the frame added. It is a \emph{length}
% value.
%
%      \begin{macrocode}
\define@key{psset}{LightThreeDLength}{%
\pst@getlength{#1}{\PstLightThreeD@XLength}%
\pst@getlength{#1}{\PstLightThreeD@YLength}}
%      \end{macrocode}
```

LightThreeSteps will be the number of steps for the light effect. It works like a resolution parameter. It is an *integer* value.

```
%    \texttt{LightThreeSteps} will be the number of steps for the
% light effect. It works like a resolution parameter. It is an
% \emph{integer} value.
%
%      \begin{macrocode}
\define@key{psset}{LightThreeDSteps}{%
\pst@getint{#1}{\PstLightThreeD@Steps}}
%      \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*.

```
%    \texttt{LightThreeDAngle} will be the angle used. It will act as
% an angle for the light effect. It is a \emph{real} value used as
% an \emph{angle}.
%
%      \begin{macrocode}
\define@key{psset}{LightThreeDAngle}{%
\pst@getangle{#1}{\PstLightThreeD@LightAngle}}
%      \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.

```
%    \texttt{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
% \texttt{setgray} operator or as the Saturation or Brightness
% component of the \texttt{sethsbcolor} one, using the HSB color
% model.
% It is a \emph{string} value.
```

```
%
%       \begin{macrocode}
\define@key{psset}{LightThreeDColorPsCommand}{%
\edef\PstLightThreeD@ColorPsCommand{#1}}
%       \end{macrocode}
```

Next, we set the default values for all these new parameters.

```
%    Next, we set the default values for all these new parameters.
%
%       \begin{macrocode}
\setkeys{psset}{%
LightThreeDXLength=0.2,LightThreeDYLength=0.3,
LightThreeDSteps=40,LightThreeDAngle=45,
LightThreeDColorPsCommand=2.5 div setgray}
%       \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.

```
% \subsection{Main macros}
%
%    The general \cs{PstLightThreeDGraphic} macro to apply the
% lighten effect on a graphic (an opened or closed curve).
% This is a PSTricks object.
%
% \begin{macro}{\PstLightThreeDGraphic}
%       \begin{macrocode}
\def\PstLightThreeDGraphic{%
\def\pst@par{}%
\pst@object{PstLightThreeDGraphic}}
%       \end{macrocode}
% \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.

```
%    Then we define it auxiliary macro which will handle the
% parameters, if some are used, initialize the PostScript
% environment and redefine the \texttt{stroke} PostScript
% operator calling two times the \texttt{LightThreeDPathForAll}
% PostScript macro defined in the header file.
%
% \begin{macro}{\PstLightThreeDGraphic@i}
%       \begin{macrocode}
\def\PstLightThreeDGraphic@i{%
\pst@makebox{%
  \PstLightThreeD@i{%
    /LightThreeDDXa LightThreeDDX def
    /LightThreeDDYa LightThreeDDY def
    /stroke {gsave reversepath LightThreeDPathForAll grestore
             LightThreeDPathForAll } def}{end}}}
%    \end{macrocode}
% \end{macro}
```

Now, the general \PstLightThreeDText macro to apply the lighten effect on a text (using PostScript fonts). This is a PSTricks object too.

```
%   Now, the general \cs{PstLightThreeDText} macro to apply the
% lighten effect on a text (using PostScript fonts). This is a
% PSTricks object too.
%
% \begin{macro}{\PstLightThreeDText}
%    \begin{macrocode}
\def\PstLightThreeDText{\def\pst@par{}\pst@object{PstLightThreeDText}}
%    \end{macrocode}
% \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 LightThreeDPathForAll PostScript macro defined in the header file, redraw a second time the outline path using the 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.

```
%   Then we define it auxiliary macro which will handle the
% parameters, if some are used, initialize the PostScript
% environment, keep the content of the \texttt{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 \texttt{show} operator. In fact,
% this code is very similar to the one of the \cs{pscharpath}
% macro from the ‘\textsf{pst-char}’ package.
%
% \begin{macro}{\PstLightThreeDText@i}
%    \begin{macrocode}
\def\PstLightThreeDText@i{%
\pst@makebox{%
  \PstLightThreeD@i{%
    /LightThreeDDXa LightThreeDDX Resolution 100 div mul def
    /LightThreeDDYa LightThreeDDY VResolution 100 div mul def
    /tx@LightThreeDSavedShow /show load def
    /show {
      dup
      gsave
        false charpath
        reversepath
        LightThreeDPathForAll
      grestore
      true charpath} def}
   {/show /tx@LightThreeDSavedShow load def
    end}%
  \begin@ClosedObj
    \def\use@pscode{%
      \pst@Verb{%
        gsave
          \tx@STV
          \pst@code
        grestore
        CP newpath moveto}%
      \gdef\pst@code{}}%
  \end@ClosedObj}}
%    \end{macrocode}
% \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.

```
% \subsection{Auxiliary macro}
%
%   Both the \cs{PstLightThreeDGraphic} and
% \cs{PstLightThreeDText} macros call the auxiliary
% \cs{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.
%
% \begin{macro}{\PstLightThreeD@i}
%    \begin{macrocode}
\def\PstLightThreeD@i#1#2{{%
%    \end{macrocode}
```

Assignment of local parameters.

```
%  Assignment of local parameters.
%
%    \begin{macrocode}
\use@par
%    \end{macrocode}
```

Initialization of the various PostScript variables which depend of the PSTricks parameters that the user can change.

```
%    Initialization of the various PostScript variables which
% depend of the PSTricks parameters that the user can change.
%
%      \begin{macrocode}
\leavevmode
\pstVerb{%
  tx@LightThreeDDict begin
    /LightThreeDDX \PstLightThreeD@XLength\space def
    /LightThreeDDY \PstLightThreeD@YLength\space neg def
    /LightThreeDSteps \PstLightThreeD@Steps def
    /LightThreeDAngle \PstLightThreeD@LightAngle def
    /LightThreeDColorPsCommand
      {\PstLightThreeD@ColorPsCommand} def
    /LightThreeDMINangle
      LightThreeDDY LightThreeDDX atan 180 sub def
    /LightThreeDMAXangle
      LightThreeDDY LightThreeDDX atan def
    #1}%
%      \end{macrocode}
```

Draw the content (text or PSTricks graphic).

```
%    Draw the content (text or PSTricks graphic).
%
%      \begin{macrocode}
\hbox{\box\pst@hbox}%
%      \end{macrocode}
```

Restoration, if needed, of some PostScript operator (for texts, the `show` one, in fact).

```
%    Restoration, if needed, of some PostScript operator
% (for texts, the “show” one, in fact).
%
%      \begin{macrocode}
\pstVerb{#2}}}
%      \end{macrocode}
% \end{macro}
```

6.5 Closing

Catcodes restoration.

```
% \subsection{Closing}
%
%    Catcodes restoration.
%
```

```
%       \begin{macrocode}
\catcode‘\@=\PstAtCode\relax
%       \end{macrocode}
```

```
%       \begin{macrocode}
%</pst-li3d>
%       \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.

```
%       \begin{macrocode}
%<*postscript-header>
%       \end{macrocode}
```

```
%       \begin{macrocode}
%!
% PostScript header file pst-li3d.pro
% Version 1.0, 2003/07/10
%
% Adapted from Peter Kleiweg <kleiweg@let.rug.nl>
% See http://odur.let.rug.nl/~kleiweg/postscript/postscript.html
%        (file depth.ps)
%
%       \end{macrocode}
```

The dictionary for this package.

```
%     The dictionary for this package.
%
%       \begin{macrocode}
/tx@LightThreeDDict 40 dict def
tx@LightThreeDDict begin
%       \end{macrocode}
```

---

[58] For explanations of them, see [51] or other references on the PostScript programming language.

[59] For 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`

```
%    The \texttt{LightThreeDMove} operator, which will replace the
% \texttt{moveto} operator in the \texttt{pathforall} call.
% Syntax is: \Verb+<x> <y> LightThreeDMove+
%
%      \begin{macrocode}
/LightThreeDMove {
   /y0c exch def
   /x0c exch def
   /xc x0c def
   /yc y0c def
   newpath
} def
%      \end{macrocode}
```

The LightThreeDLine operator, which will replace the lineto operator in the pathforall call. Syntax is: `<x> <y> LightThreeDLine`

```
%    The \texttt{LightThreeDLine} operator, which will replace the
% \texttt{lineto} operator in the \texttt{pathforall} call.
% Syntax is: \Verb+<x> <y> LightThreeDLine+
%
%      \begin{macrocode}
/LightThreeDLine {
   /yyc exch def
   /xxc exch def
   yyc yc sub
   xxc xc sub
   1 index 0 eq 1 index 0 eq and not {
      atan
      /ac exch def
      ac LightThreeDMINangle le
          ac LightThreeDMAXangle ge or {
         ac LightThreeDAngle sub
         2 mul
         cos
         1 add
         LightThreeDColorPsCommand
         xc yc moveto
         xxc yyc lineto
         LightThreeDDXa LightThreeDDYa rlineto
         xc LightThreeDDXa add yc LightThreeDDYa add lineto
         closepath
         fill
      } if
   } if
   /xc xxc def
   /yc yyc def
} def
```

```
%       \end{macrocode}
```

The LightThreeDCurve operator, which will replace the curveto operator in the pathforall call. Syntax is:

`<x2> <y2> <x3> <y3> <x4> <y4> LightThreeDCurve`

```
%     The \texttt{LightThreeDCurve} operator, which will replace the
% \texttt{curveto} operator in the \texttt{pathforall} call.
% Syntax is:
% \Verb+<x2> <y2> <x3> <y3> <x4> <y4> LightThreeDCurve+
%
%       \begin{macrocode}
/LightThreeDCurve {
   /y4c exch def
   /x4c exch def
   /y3c exch def
   /x3c exch def
   /y2c exch def
   /x2c exch def
   /y1c yc def
   /x1c xc def
%       \end{macrocode}
```

It is faster to make some preliminary computations outside the loop, as in the original file from Peter Kleiweg, but with TEX it can generate huge numbers which make the PostScript interpreter crash![60]

```
%     It is faster to make some preliminary computations outside
% the loop, as in the original file form Peter Kleiweg, but it
% can generate huge numbers which make the PostScript
% interpreter crash!\footnote{%
% It still can occur on some (unknown) circumstances. In such
% cases, just moving the text or graphinc is generally enough
% to cure the problem...}
%
%       \begin{macrocode}
% Computations order changed from the original file
   1 LightThreeDSteps div 1 LightThreeDSteps div 1 {
         /t exch def
         3 t sub x1c mul
         t 2 sub x2c mul
            1 t sub x3c mul add 3 mul add
         x4c t mul add t mul
         x2c x1c sub 3 mul add t mul
         x1c add% X
         3 t sub y1c mul
         t 2 sub y2c mul
```

---

[60] It still can occur on some (unknown) circumstances. In such cases, just moving the text or graphic is generally enough to cure the problem...

```
          1 t sub y3c mul add 3 mul add
        y4c t mul add t mul
        y2c y1c sub 3 mul add t mul
        y1c add% Y
        LightThreeDLine
  } for
} def
%      \end{macrocode}
```

The LightThreeDClose operator, which will replace the closepath operator in the pathforall call. Syntax is: LightThreeDClose

DG: Code modified because the original one crash the PostScript interpreter of two printers that I tested, even if GhostScript handle it correctly...

```
%    The \texttt{LightThreeDClose} operator, which will replace
% the \texttt{closepath} operator in the \texttt{pathforall} call.
% Syntax is: \Verb+LightThreeDClose+
%
%      \begin{macrocode}
/LightThreeDClose {
%  x0c y0c LightThreeDLine% Anomaly on some printers!
  x0c 0 eq {x0c} {x0c 1 add} ifelse
        y0c 0 eq {y0c} {y0c 1 add} ifelse LightThreeDLine
  newpath
} def
%      \end{macrocode}
```

The LightThreeDPathForAll operator, which will replace the pathforall operator to draw the current path with a three dimensional lighten effect.

```
%    The \texttt{LightThreeDPathForAll} operator, which will
% replace the \texttt{pathforall} operator to draw the current
% path with a three dimensional lighten effect.
%
%      \begin{macrocode}
/LightThreeDPathForAll {
  { LightThreeDMove} { LightThreeDLine }
    { LightThreeDCurve } { LightThreeDClose } pathforall} def
%      \end{macrocode}
```

Then, to finish, we close the dictionary.

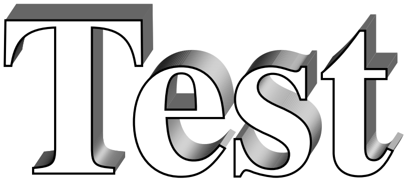
```
%    Then, to finish, we close the dictionary.
%
%      \begin{macrocode}
end
%      \end{macrocode}
```

```
%      \begin{macrocode}
%</postscript-header>
%      \end{macrocode}
```

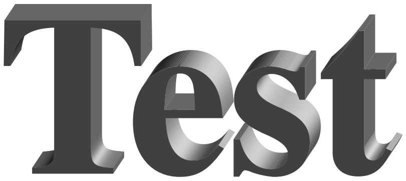
And, at the end, we print the index and the list of changes of the package.

```
% \Finale
% \PrintIndex
% \PrintChanges
%
\endinput
%%
%% End of file ‘pst-li3d.dtx’
```

Now, the new package can be used!

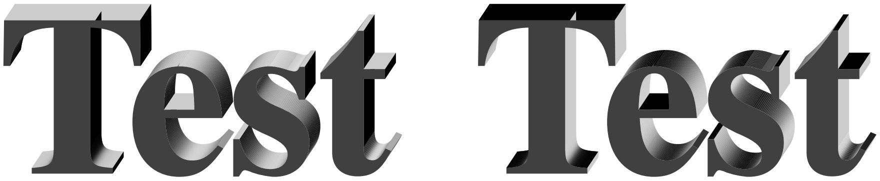


```
\DeclareFixedFont{\Bf}{T1}{ptm}{b}{n}{3cm}
\PstLightThreeDText[fillstyle=solid,fillcolor=white]{\Bf Test}
```

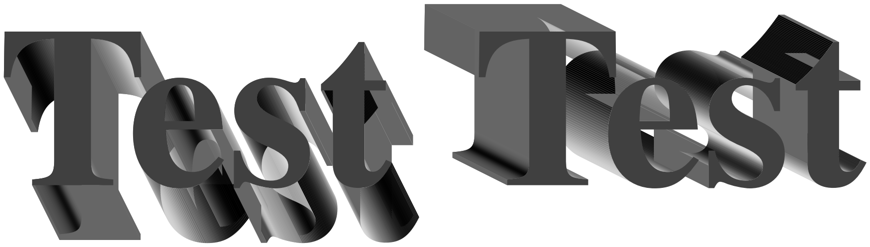


```
\PstLightThreeDText[linestyle=none,fillstyle=solid,
    fillcolor=darkgray]{\Bf Test}
```

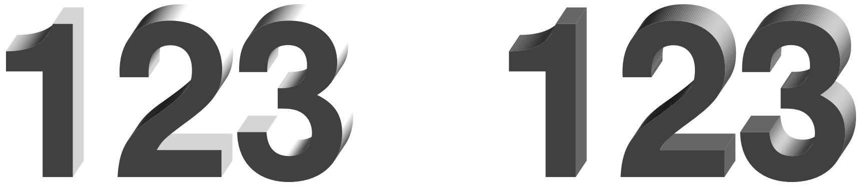
```
\psset{linestyle=none,fillstyle=solid,fillcolor=darkgray}%
\PstLightThreeDText[LightThreeDAngle=0]{\Bf Test}
\hfill
\PstLightThreeDText[LightThreeDAngle=90]{\Bf Test}
```



```
\psset{linestyle=none,fillstyle=solid,fillcolor=darkgray}%
\PstLightThreeDText[LightThreeDXLength=0.5,
    LightThreeDYLength=-1]{\Bf Test}
\hfill
\PstLightThreeDText[LightThreeDXLength=-1,
    LightThreeDYLength=0.5]{\Bf Test}
```

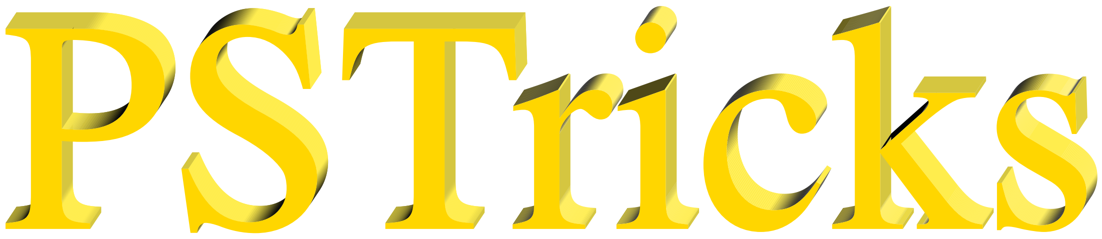


```
\DeclareFixedFont{\Sf}{T1}{phv}{b}{n}{3cm}
\psset{linestyle=none,fillstyle=solid,fillcolor=darkgray}%
\PstLightThreeDText[LightThreeDColorPsCommand=%
   1.2 div setgray]{\Sf 123}
\hfill
\PstLightThreeDText[LightThreeDColorPsCommand=%
   2.5 div setgray]{\Sf 123}
```



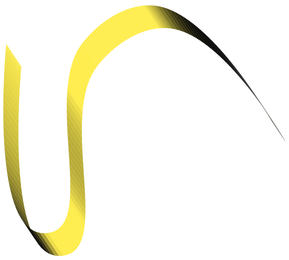
```
\DeclareFixedFont{\Rm}{T1}{ptm}{m}{n}{3cm}
\psset{linestyle=none,fillstyle=solid}%
\PstLightThreeDText[fillcolor=Violet,
      LightThreeDColorPsCommand=%
         2.5 div 0.7 exch 0.8 sethsbcolor]{%
   \Rm 987}
\hfill
\PstLightThreeDText[fillcolor=DarkGreen,
      LightThreeDColorPsCommand=%
         2 div 0.5 exch 0.2 exch sethsbcolor]{%
   \Rm 987}
```

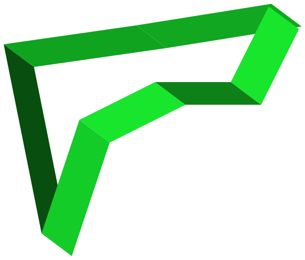
```
\DeclareFixedFont{\Rmb}{T1}{ptm}{m}{n}{4cm}
\PstLightThreeDText[linestyle=none,fillstyle=solid,fillcolor=Gold,
      LightThreeDColorPsCommand=%
         1.2 div 0.15 exch 0.7 exch sethsbcolor]{%
   \Rmb PSTricks}
```

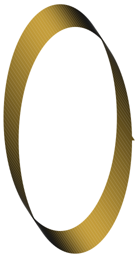


```
\psset{unit=0.5}%
\pspicture(-0.1,-3.5)(7.2,3)
```

```
  \PstLightThreeDGraphic[LightThreeDXLength=0.4,
      LightThreeDColorPsCommand=%
       1.2 div 0.15 exch 0.7 exch sethsbcolor]{%
    \pscurve(0,2)(1,-3)(2,2)(4,3)(7,0)}
\endpspicture
\hfill
\pspicture(0,-3.5)(7.7,3)
  \PstLightThreeDGraphic[LightThreeDXLength=0.8,
      LightThreeDColorPsCommand=%
        2 div 0.35 exch 0.9 exch sethsbcolor]{%
    \pspolygon(0,2)(1,-3)(2,0)(4,1)(6,1)(7,3)}
\endpspicture
\hfill
\pspicture(0.5,-3.6)(3.8,3)
  \PstLightThreeDGraphic[LightThreeDColorPsCommand=%
      2.6 div 0.12 exch 0.7 exch sethsbcolor]{%
    \psellipse(2,0)(1.5,3)}
\endpspicture
```

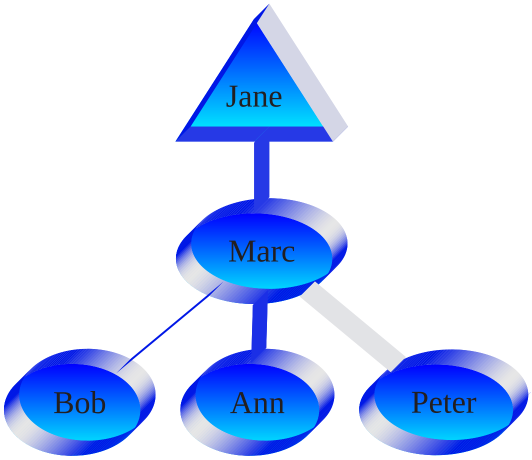






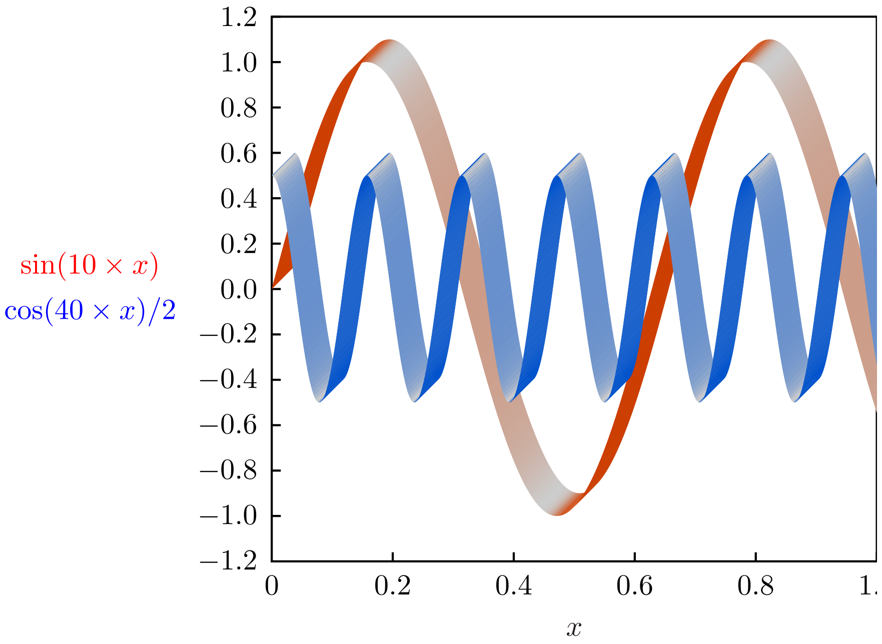
Here we draw an ellipse. The second case is more sophisticated, as we will increase continously the starting value of the Brightness component of the HSB color, using a PostScript variable.

```
\SpecialCoor
\def\PstCoordinates{}%
\Multido{\nDistance=0.00+0.02,\iAngle=0+20}{200}{%
  \edef\PstCoordinates{\PstCoordinates(\nDistance;\iAngle)}}
\psset{unit=0.5}%
\pspicture(-3.8,-4)(4.1,3.7)
  \PstLightThreeDGraphic[LightThreeDLength=0.2,
      LightThreeDColorPsCommand=%
        1.2 div 0.3 exch 0.7 exch sethsbcolor]{%
    \expandafter\pscurve\PstCoordinates}
\endpspicture
\hfill
\pspicture(-3.8,-4)(4.1,3.7)
  \PstLightThreeDGraphic[LightThreeDLength=0.2,
      LightThreeDAngle=30,LightThreeDColorPsCommand=%
        /Counter Counter 0.00005 add def
        2 mul Counter exch 0.7 exch sethsbcolor]{%
    \pstVerb{/Counter 0 def}%
    \expandafter\pscurve\PstCoordinates}
\endpspicture
```



```
\PstLightThreeDGraphic[LightThreeDXLength=0.2,
        LightThreeDYLength=-0.2,
        LightThreeDColorPsCommand=%
          1.2 div 0.65 exch 0.9 sethsbcolor]{%
    \large
    \let\TovalORIG\Toval
    \def\Toval#1{\TovalORIG{\raise2mm\hbox{\hskip2mm#1}}}%
    \let\TtriORIG\Ttri
    \def\Ttri#1{\TtriORIG{\raise3mm\hbox{#1}}}%
    \psset{framesep=0.15,fillstyle=gradient,gradmidpoint=0,
              gradbegin=cyan,gradend=blue}%
    \pstree[treesep=0.5]{\Ttri{Jane}}
        {\psset{framesep=0.25}%
          \pstree{\Toval{Marc}}
              {\Toval{Bob}\Toval{Ann}\Toval{Peter}}}}
```

```
\psset{xunit=8,yunit=3}%
\pspicture*(-0.45,-1.6)(1,1.3)
    \psaxes[Dx=0.2,Oy=-1.2,Dy=0.2,tickstyle=top,
                axesstyle=frame](0,-1.2)(1,1.2)
    \rput(-0.3,0.1){\textcolor{red}{$\sin (10 \times x)$}}
    \rput(-0.3,-0.1){\textcolor{blue}{$\cos (40 \times x) / 2$}}
    \rput(0.5,-1.5){$x$}
    \psset{plotpoints=500,LightThreeDXLength=0.3,
        LightThreeDYLength=-0.3}%
    \PstLightThreeDGraphic[LightThreeDColorPsCommand=%
            1.5 div 0.05 exch 0.8 sethsbcolor]{%
        \psplot{0}{1}{x 10 mul 57.296 mul sin}}% sin(10 x)
    \PstLightThreeDGraphic[LightThreeDColorPsCommand=%
            1.5 div 0.6 exch 0.8 sethsbcolor]{%
        \psplot{0}{1}{x 40 mul 57.296 mul cos 2 div}}% cos(40 x) / 2
\endpspicture
```


sin(10 × x)
cos(40 × x)/2
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
0
0.2
0.4
0.6
0.8
1.
x

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:

```
\newgray{darkgray}{.25}
```

---

[61] However, this is not necessary with the PSTricks LR-box commands, expect 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,

```
\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,

```
\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,

```
\newcmykcolor{hercolor}{.5 1 0 .5}
```

For defining new colors, the *rbg* 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

```
\Lput{labelsep}[refpoint]{rotation}(pos){stuff}
```

It is a combination of **\Rput** and **\lput**, equivalent to

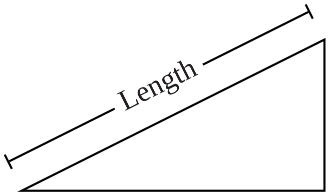
```
\lput(pos){\Rput{labelsep}[refpoint]{rotation}(0,0){stuff}}
```

\Mput is a short version of **\Lput** with no {<rotation>} or (<pos>) argument. **\Lput** and **\Mput** remain part of PSTricks only for backwards compatibility.

Here are the node label commands:

\lput*[*refpoint*]{*rotation*}**(*pos*){*stuff*}**

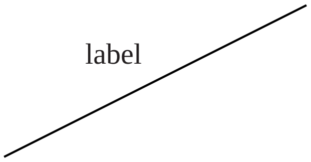
The l stands for “label”. Here is an example illustrating the use of the optional star and :<angle> with **\lput**, as well as the use of the **offset** parameter with **\pcline**:



```
\pspolygon(0,0)(4,2)(4,0)
\pcline[offset=12pt]{|-|}(0,0)(4,2)
\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.,

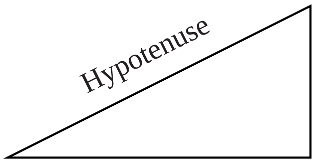


```
\pcline(0,0)(4,2)
\lput{:U}{\rput[r]{N}(0,.4){label}}
```

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 **\aput** 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 **\uput**. For example:



```
\pspolygon(0,0)(4,2)(4,0)
\pcline[linestyle=none](0,0)(4,2)
\aput{:U}{Hypotenuse}
```

---

[62] There 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.

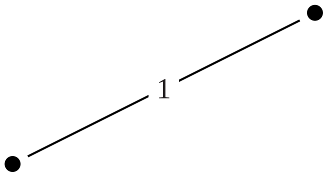
\bput*[*labelsep*]{*angle*}***(pos){stuff}***

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:

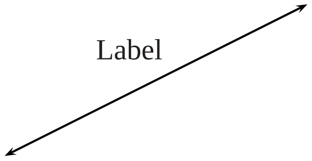
\mput*[*refpoint*]***{stuff}***
\Aput*[*labelsep*]***{stuff}***
\Bput*[*labelsep*]***{stuff}***

of **\lput**, **\aput** and **\bput**, respectively, that have no angle or positioning argument. For example:



```
\cnode*(0,0){3pt}{A}
\cnode*(4,2){3pt}{B}
\ncline[nodesep=3pt]{A}{B}
\mput*{1}
```

Here is another:



```
\pcline{<->}(0,0)(4,2)
\Aput{Label}
```

10 Coordinates commands

There is an obsolete command **\Polar** that causes coordinates in the form (<r>,<a>) 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.,

```
\Cartesian(x,y)
```

has the effect of

```
\psset{xunit=x,yunit=y}
```

\Cartesian can be used for this purpose without using **\Polar**.

Help[63]

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 it 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 made his own choice among the various solutions available.

---

[63] This 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. Malyshev and VTEX [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 VTEX 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 PdfTEX workarounds

The PdfTEX 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 PdfTEX 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 PdfTEX 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 LATEX, 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

---

[64] For 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:

```
\documentclass{article}

\usepackage{pdftricks}

\begin{psinputs}
  \usepackage{times}
  \usepackage{pstcol}
  \usepackage{pst-plot}
\end{psinputs}

\begin{document}

\begin{pdfpic}
  \psset{xunit=8,yunit=3}%
  \begin{pspicture}(-0.45,-1.6)(1.05,1.3)
    \psaxes[Dx=0.2,Oy=-1.2,Dy=0.2,tickstyle=top,
           axesstyle=frame](0,-1.2)(1,1.2)
  \rput(-0.3,0){%
    \shortstack{\textcolor{red}{$\sin (10 \times x)$}\\
              \textcolor{blue}{$\cos (40 \times x) / 2$}}}
    \rput(-0.3,0){%
      \shortstack{\textcolor{red}{$\sin (10 \times x)$}\\
                \textcolor{blue}{$\cos (40 \times x) / 2$}}}
    \rput(0.5,-1.5){\shortstack{$x$\\Sample plots}}
    \psplot[linecolor=red]% 1 radian = 57.296 degrees
      {0}{1}{x 10 mul 57.296 mul sin}% sin(10 x)
    \psplot[linecolor=blue,plotpoints=200]
      {0}{1}{x 40 mul 57.296 mul cos 2 div}% cos(40 x) / 2
  \end{pspicture}
\end{pdfpic}

\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):

```
pdflatex -shell-escape FileName
```

On it side, ps4pdf require a PostScript to PDF converter like GhostScript. It also need the ‘preview’ package [52] from David Kastrup. With ps4pdf,

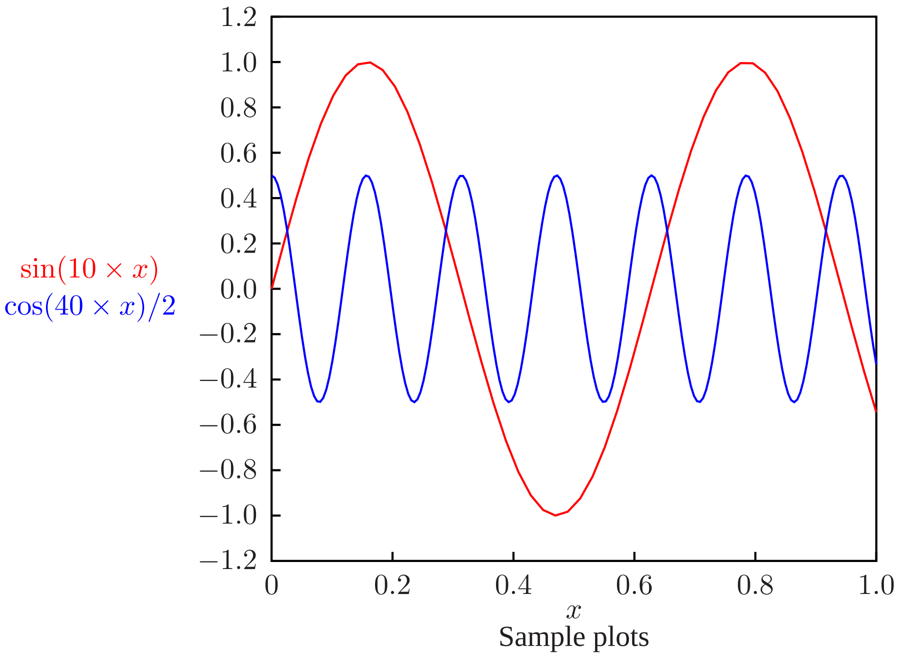


Figure 4: Output of the example about pdfTricks and ps4pdf usage

the preceding example must look like:

```
\documentclass{article}

\usepackage{graphicx}
\usepackage{ps4pdf}

\PSforPDF{%
  \usepackage{pstcol}
  \usepackage{pst-plot}}

\begin{document}

\PSforPDF{%
  \psset{xunit=8,yunit=3}%
  \begin{pspicture}(-0.45,-1.6)(1.05,1.3)
    \psaxes[Dx=0.2,Oy=-1.2,Dy=0.2,tickstyle=top,
           axesstyle=frame](0,-1.2)(1,1.2)
    \rput(-0.3,0){%
      \shortstack{\textcolor{red}{$\sin  (10  \times  x)$}\\
                \textcolor{blue}{$\cos  (40  \times  x)  /  2$}}}
    \rput(0.5,-1.5){\shortstack{$x$\\Sample  plots}}
    \psplot[linecolor=red]%  1  radian  =  57.296  degrees
      {0}{1}{x  10  mul  57.296  mul  sin}%  sin(10  x)
    \psplot[linecolor=blue,plotpoints=200]
```

```
        {0}{1}{x 40 mul 57.296 mul cos 2 div}% cos(40 x) / 2
    \end{pspicture}}

\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):

```
latex FileName
dvips -Ppdf -G0 -o FileName-pics.ps FileName.dvi
ps2pdf FileName-pics.ps
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.)

```
\define@key{psset}{Amortization}{%
   \edef\PstWavesInterferences@Amortization{#1}}
\define@key{psset}{Amplitude1Coeff}{%
   \edef\PstWavesInterferences@AmplitudeACoeff{#1}}
\define@key{psset}{Amplitude2Coeff}{%
   \edef\PstWavesInterferences@AmplitudeBCoeff{#1}}
\define@key{psset}{Celerity}{%
   \edef\PstWavesInterferences@Celerity{#1}}
\define@key{psset}{Frequency1}{%
   \edef\PstWavesInterferences@FrequencyA{#1}}
\define@key{psset}{Frequency2}{%
   \edef\PstWavesInterferences@FrequencyB{#1}}
\define@key{psset}{Phase1}{%
   \edef\PstWavesInterferences@PhaseA{#1}}
\define@key{psset}{Phase2}{%
   \edef\PstWavesInterferences@PhaseB{#1}}
\define@key{psset}{Resolution}{%
   \edef\PstWavesInterferences@Resolution{#1}}
\define@key{psset}{Time}{\edef\PstWavesInterferences@Time{#1}}
\define@key{psset}{X1}{%
   \edef\PstWavesInterferences@SourceXA{#1}}
\define@key{psset}{X2}{%
   \edef\PstWavesInterferences@SourceXB{#1}}
\define@key{psset}{Y1}{%
   \edef\PstWavesInterferences@SourceYA{#1}}
```

```
\define@key{psset}{Y2}{%
  \edef\PstWavesInterferences@SourceYB{#1}}

\setkeys{psset}{Amortization=5,Amplitude1Coeff=1,Amplitude2Coeff=1,
  Celerity=0.20,Frequency1=20,Frequency2=20,Phase1=0,Phase2=0,
  Resolution=1,Time=0.20,X1=-2,X2=2,Y1=0,Y2=0}

\def\PstWavesInterferences{%
\def\pst@par{}\pst@object{PstWavesInterferences}}

\def\PstWavesInterferences@i(#1,#2)(#3,#4){%
\pspicture(#1,#2)(#3,#4)
\begin@AltOpenObj
\addto@pscode{%
  /AmplitudeMax 0.5 def
  #1 28.45 mul \PstWavesInterferences@Resolution\space
               #3 28.45 mul { % Loop on abscissas
   /abscissept exch def % In points
   /abscisse abscissept 2845 div def % In meters
   #2 28.45 mul \PstWavesInterferences@Resolution\space
                #4 28.45 mul { % Loop on ordinates
    /ordonneept exch def % In points
    /ordonnee ordonneept 2845 div def % In meters
    /d1 abscisse \PstWavesInterferences@SourceXA\space
          100 div sub dup mul
        ordonnee \PstWavesInterferences@SourceYA\space
          100 div sub dup mul add sqrt def
    /d2 abscisse \PstWavesInterferences@SourceXB\space
          100 div sub dup mul
        ordonnee \PstWavesInterferences@SourceYB\space
          100 div sub dup mul add sqrt def
    /yS1 360 \PstWavesInterferences@FrequencyA\space mul
      \PstWavesInterferences@Time\space d1
      \PstWavesInterferences@Celerity\space div sub mul
      \PstWavesInterferences@PhaseA\space add sin
      AmplitudeMax 2.71828 d1 neg
      \PstWavesInterferences@Amortization\space mul exp mul
      \PstWavesInterferences@AmplitudeACoeff\space mul mul def
    /yS2 360 \PstWavesInterferences@FrequencyB\space mul
      \PstWavesInterferences@Time\space d2
      \PstWavesInterferences@Celerity\space div sub mul
      \PstWavesInterferences@PhaseB\space add sin
      AmplitudeMax 2.71828 d2 neg
      \PstWavesInterferences@Amortization\space mul exp mul
      \PstWavesInterferences@AmplitudeBCoeff\space mul mul def
    yS1 0.5 add yS2 0.5 add 1 setrgbcolor % Color
    newpath
    abscissept ordonneept
      \PstWavesInterferences@Resolution\space 0 360 arc
    closepath
    fill
   } for
  } for}%
\end@OpenObj
\endpspicture}

% Generation of the images in EPS format
\multido{\nTime=0.20+0.01}{3}{%
  % As all file names must have the same number of characters,
  % we must add a 0 for the 9 first ones if we have between
  % 10 and 99 images
  \ifnum\multidocount<10\relax\def\Pad{0}\else\def\Pad{}\fi
  \PSTtoEPS[headerfile=pstricks.pro,headers=user,
             bbllx=-8,bblly=-4,bburx=8,bbury=4]
            {WavesInterferences\Pad\the\multidocount.eps}{%
    \PstWavesInterferences[Amortization=10,Celerity=0.2,
      Time=\nTime,Frequency1=20,Frequency2=20,Resolution=3,
      X1=-4,X2=4,Y1=-3,Y2=-3](-8,-4)(8,4)}}
```

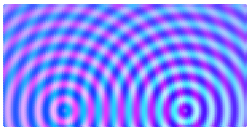


Image 1

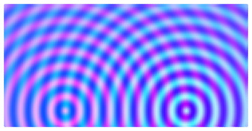


Image 2

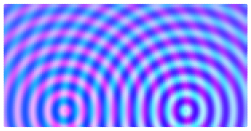


Image 3

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.

```
#! /bin/sh

set -x

# Generation of an animated GIF file
# from PSTricks pure graphics

  if [ $# != 2 ]
   then echo Syntax: PstAnimatedPureGraphics.sh \
                        file_without_suffix delay
          exit
fi

latex $1

for file1 in $1?*.eps
```

```
do
   file2=‘basename $file1 .eps‘
   convert $file1 $file2.gif # Convert each file in GIF format
done

# Generate the animated GIF file using “gifsicle”
gifsicle –delay $2 –loop –colors 256 –output $1.gif $1?*.gif

# We can also use “convert” from ImageMagick
# (but it could produce in some cases huge files)
# convert -delay $2 -loop 0 -colors 256 $1?*.gif $1.gif

# Clean up temporary files
rm -f $1?*.{eps,gif} $1.{aux,dvi,log}

animate $1.gif & # From ImageMagick, or “gifview -a $1.gif”

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):

```
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.

```
\documentclass{minimal}

\usepackage{pst-eps}
\usepackage{pst-eucl}

% Utility macro to generate intermediate images,
% running parts of the complete code of a macro.
% Code #3 will be included if #1 >= #2.
\def\IncludeCodeForImage#1#2#3{%
\ifnum#2>#1\relax
\else
```

```
   #3
\fi}

\def\PstOrthocentre{%
\typein[\Image]{^^JWhich image?}
\TeXtoEPS
   \pspicture(-1.5,-1.3)(3.6,2.2)
      \psset{CodeFig=true}%
      \IncludeCodeForImage{\Image}{1}{%
         \pstTriangle[PosAngleA=180](-1,0){A}(3,-1){B}(3,2){C}}
      \IncludeCodeForImage{\Image}{2}{%
         \pstProjection[PosAngle=-90]{B}{A}{C}{C’}}
      \IncludeCodeForImage{\Image}{3}{%
         \pstProjection{B}{C}{A}{A’}}
      \IncludeCodeForImage{\Image}{4}{%
         \pstProjection[PosAngle=90]{A}{C}{B}{B’}}
      \IncludeCodeForImage{\Image}{5}{%
         \pstInterLL[PosAngle=135,PointSymbol=square]
                        {A}{A’}{B}{B’}{H}}
   \endpspicture
\endTeXtoEPS

\begin{document}
   \PstOrthocentre
\end{document}
```

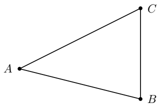


Image 1

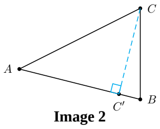


Image 2

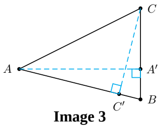


Image 3

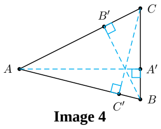


Image 4

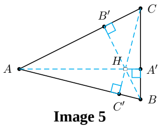


Image 5

Here we give an example script for Unix systems which allow to automatize all the process.

```
#! /bin/sh

set -x

# Generation of an animated GIF file
# from PSTricks non pure graphics

if [ $# != 3 ]
   then echo Syntax: PstAnimatedNonPureGraphics.sh \
                   file_without_suffix number_images delay
        exit

declare -i i
```

```

i=1
while [ $i -le $2 ]
do
  echo $i | latex Orthocentre
  if [ $i -lt 10 ]
     then pad=“00”
     else if [ $i -lt 100 ]
             then pad=“0”
             else pad=“”
          fi
  fi
  dvips -E -o $1$pad$i.eps $1.dvi
  # Convert the file in GIF format
  convert $1$pad$i.eps $1$pad$i.gif
  i=$i+1
done

# Generate the animated GIF file using “gifsicle”
gifsicle –delay 200 –loop –colors 256 \
       –output $1.gif $1?*.gif

# We can also use “convert” from ImageMagick
# (but it could produce in some cases huge files)
# convert -delay 200 -loop 0 -colors 256 $1?*.gif $1.gif

# Clean up temporary files
rm -f $1?*.{eps,gif} $1.{aux,dvi,log}

animate $1.gif & # From ImageMagick, or “gifview -a $1.gif”

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):

```
PstAnimatedNonPureGraphics.sh Orthocentre 5 200
```

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.

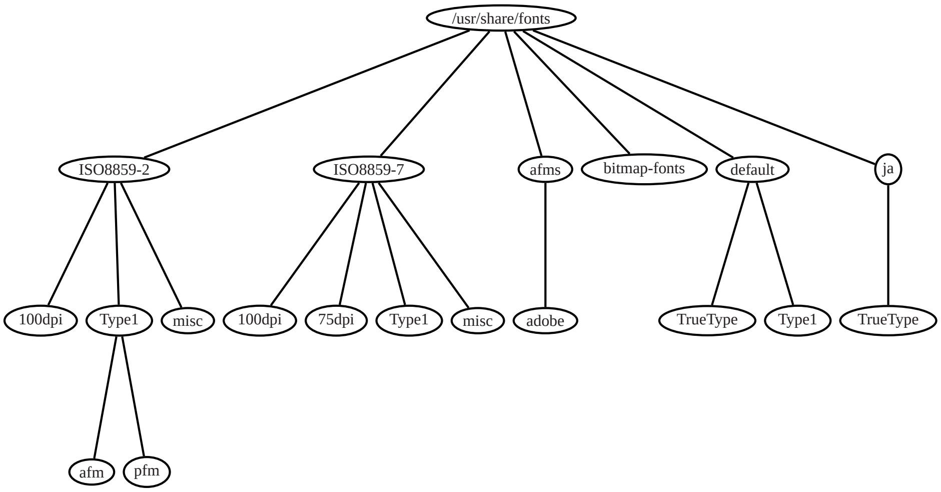
```
#! /usr/bin/perl

use Getopt::Long;

$TreeDirectory = “.”;# Default directory is the current one
$RecursionLevelMax = 10;# Better than infinity!

&OptionsRead;
if (! -d $TreeDirectory) {
    print “$TreeDirectory is not a directory!\n”;
    exit 0
}
print “\\pstree{\\PstTreeDirectory{$TreeDirectory}}\n {“;
&ListSubdirectories($TreeDirectory, 0);
print “\n”;
exit 0;

# To get list of subdirectories (recursive subprogram)
sub ListSubdirectories {
    local ($directory, $recursion_level) = @_;
    local ($boolean, $dir_temp, $dir_print_temp, @dir, @subdir,
            *DIR);

    opendir (DIR, $directory) or return;
        @dir = grep {!/\/\.\.?$/} grep { -d } map {“$directory/$_”}
                readdir (DIR);
    closedir (DIR);

    $recursion_level++;

    if ($recursion_level <= $RecursionLevelMax) {
        $boolean = 0;
        foreach $dir_temp (sort @dir) {# Loop on subdirectories
            # For special characters that are a problem for TeX
            $dir_print_temp = $dir_temp;
            $dir_print_temp =~ s/.*\///, s/[\$,\&,%,\#,\_,\{,\}]/\\\_/g;

            if ($boolean != 0) {print “\n”, “ “ x (2*$recursion_level)}

            # We must see if this directory has subdirectories
```

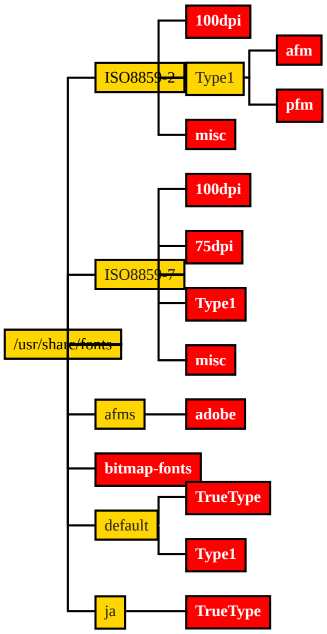
```
      opendir (DIR, “$dir_temp”);
      @subdir = grep {!/\/\.\.?$/} grep { -d }
                 map {“$dir_temp/$_”} readdir (DIR);
      closedir (DIR);

      if ($#subdir != -1 &&
          $recursion_level != $RecursionLevelMax) {
        # At least one subdirectory
        print “\\pstree{\\PstTreeDirectory{$dir_print_temp}}\n”,
          “ “ x (2 * $recursion_level + 1), “{“;
        }
      else {
        print “\\PstTreeDirectory{$dir_print_temp}”;
      }

      $boolean = 1;
      &ListSubdirectories(“$dir_temp”, $recursion_level);
    }

    if ($boolean != 0) {print “}”}
  }
}

# Reading options
sub OptionsRead{
  GetOptions (qw (-d=s -rl=i));# Option definitions

  # Option analysis
  if ($opt_d) {$TreeDirectory = $opt_d};
  if ($opt_rl) {$RecursionLevelMax = $opt_rl};
}
```

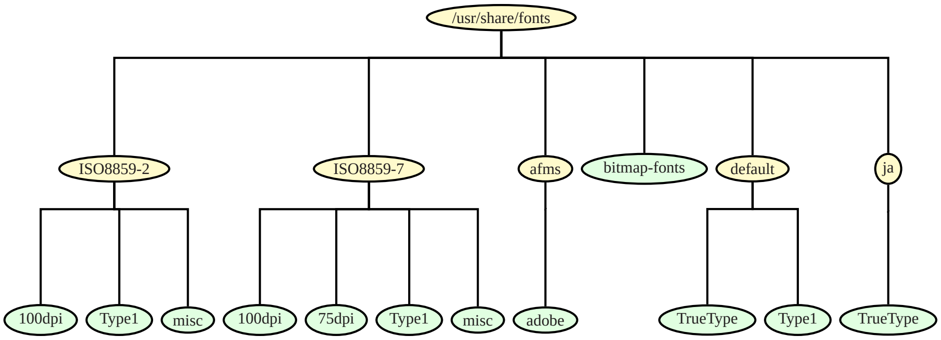
and here are some examples of usage, with various customizations. The commands given which have generated the files included are shown in the comments.

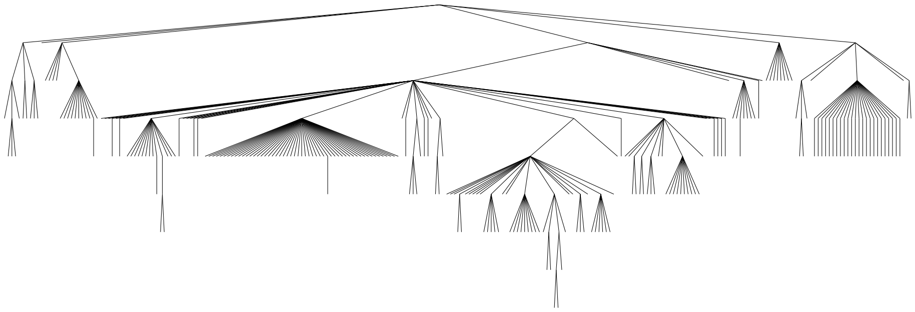
```
\tiny

% pst-dir.pl -d /usr/share/fonts >dirtree1.tex
\bgroup
  \def\PstTreeDirectory#1{\Toval[framesep=0.05]{#1}}%
  \psset{treesep=0.1}%
  \input{dirtree1.tex}
\egroup

\bgroup
  \def\PstTreeDirectory#1{%
  \ifx\pstree@next\relax
```

```
    \def\pst@tempa{\psframebox[fillstyle=solid,fillcolor=Gold]{#1}}%
  \else
    \def\pst@tempa{\psframebox[fillstyle=solid,fillcolor=red]{%
                        \textcolor{white}{\textbf{#1}}}}%
  \fi
  \TR{\pst@tempa}}%
  \def\psedge{\ncangle[angleB=180]}% Horizontal edges
  \psset{treemode=R,treesep=0.3,href=-1,levelsep=*1.2}%
  \input{dirtree1.tex}
\egroup

% pst-dir.pl -d /usr/share/fonts -rl 2 >dirtree2.tex
\bgroup
  \psset{treesep=0.1}%
  \def\PstTreeDirectory#1{%
  \ifx\pstree@next\relax
    \def\pst@tempa{LemonChiffon}%
  \else
    \def\pst@tempa{PaleGreen}%
  \fi
  \Toval[framesep=0.05,fillstyle=solid,fillcolor=\pst@tempa]{#1}}%
  \def\psedge{\ncangles[angleA=-90,angleB=90]}% Horizontal edges
  \input{dirtree2.tex}
\egroup

% pst-dir.pl -d /usr/X11R6 >dirtree3.tex
\bgroup
  \psset{treefit=loose,treesep=0.2}%
  \def\PstTreeDirectory#1{\Tp}%
  \scaleboxto(12,4){\input{dirtree3.tex}}
\egroup
```









D Loops

Loops are a key basic construction in all programs and are often required also to buid computational graphics. The macros **\multirput**, to place multiple copies of texts or graphics at equally spaced positions expressed in Cartesian coordinates, and **\multips** 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’ package[65] [45] had already been shown in various preceding chapters, but this is important to emphasize it various features, as it usage is common.[66]

Refer to it 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 put it body inside a group, at the opposite of **\Multido**,
- the **\mmultido** and **\MMultido** variants increment the counters before starting,
- the **\multidostop** macro allow to interrupt a loop,
- the **\multidocount** counter store the value of the iteration number.

We give here some other examples, to illustrate the various usages of such loops. You will still found 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:

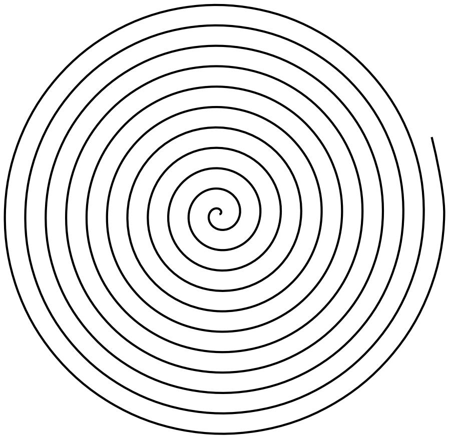
- the straightforward way to draw a spiral is to use polar plots (see Section L.15),
- the technique used here to compute coordinates, to store them and reuse them later is very general,
- we use here the **\Multido** variant; if we would have used the **\multido** one, we must have used the \xdef macro 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.

---

[65] You can also use the ‘repeat’ package [74] from Victor Eijkhout or, if you use only LaTeX, the \whiledo macro of the ‘ifthen’ package [31] or the internal LaTeX macros \@for, \@whilenum, etc.

[66] Nevertheless, in the restricted cases where it is applicable, the **\multips** and **\multirput** macros are more efficient and produce more compact code than the **\multido** one.

[67] Usage of *reals* rather than *numbers* is faster but less precise, with possible rounding approximations during computations.



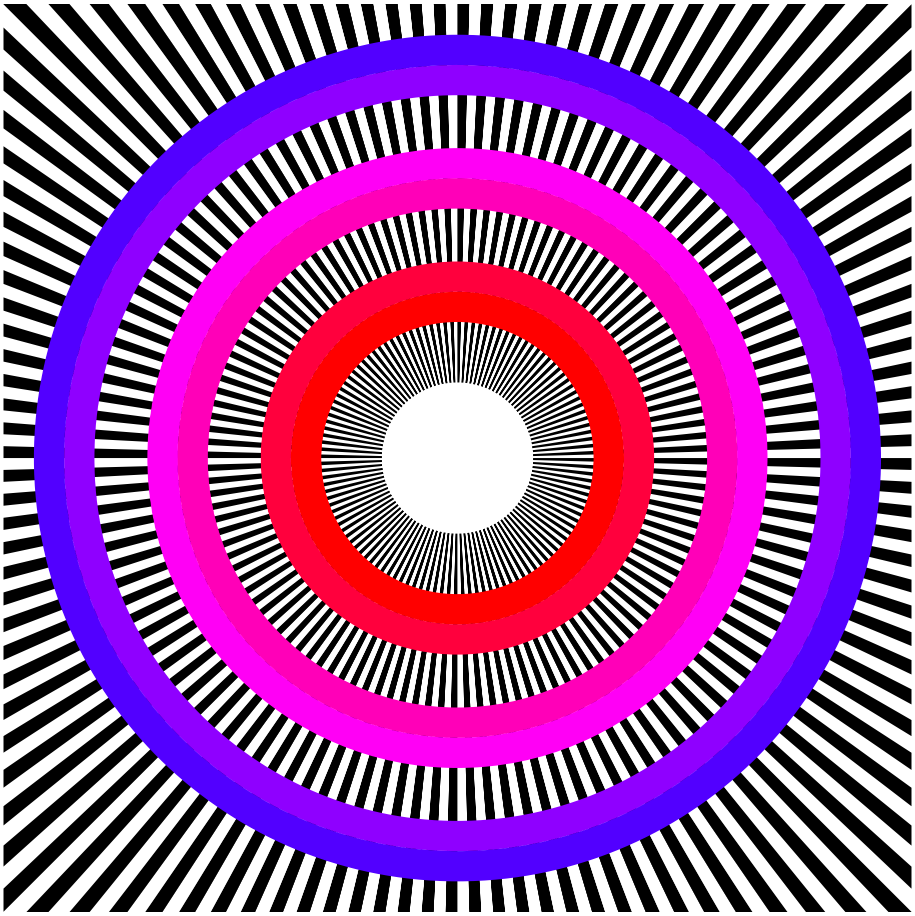
```
\SpecialCoor
\psset{unit=0.75}%
\def\PstCoordinates{}%
\Multido{\nDistance=0.00+0.02,\iAngle=0+20}{200}{%
  \edef\PstCoordinates{\PstCoordinates(\nDistance;\iAngle)}}
\expandafter\pscurve\PstCoordinates
```

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.

```
\pspicture*(-6,-6)(6,6)
  \degrees[240]
  \SpecialCoor
  \multido{\iAngleA=1+2,\iAngleB=2+2}{120}{%
    \pspolygon*(1;\iAngleA)(9;\iAngleA)(9;\iAngleB)(1;\iAngleB)}
  \multido{\nHueA=1.00+-0.12,\nRadiusA=2.0+1.5}{3}{%
    \multido{\nHueB=\nHueA+-0.04,\nRadiusB=\nRadiusA+0.4}{2}{%
      \definecolor{MyColor}{hsb}{\nHueB,1,1}%
      \pscircle[linewidth=0.4,linecolor=MyColor]{\nRadiusB}}}
\endpspicture
```

---

[68] Look at it fixedly for few seconds, and you will see the colored circles turning fast. Of course, this is an optical illusion...



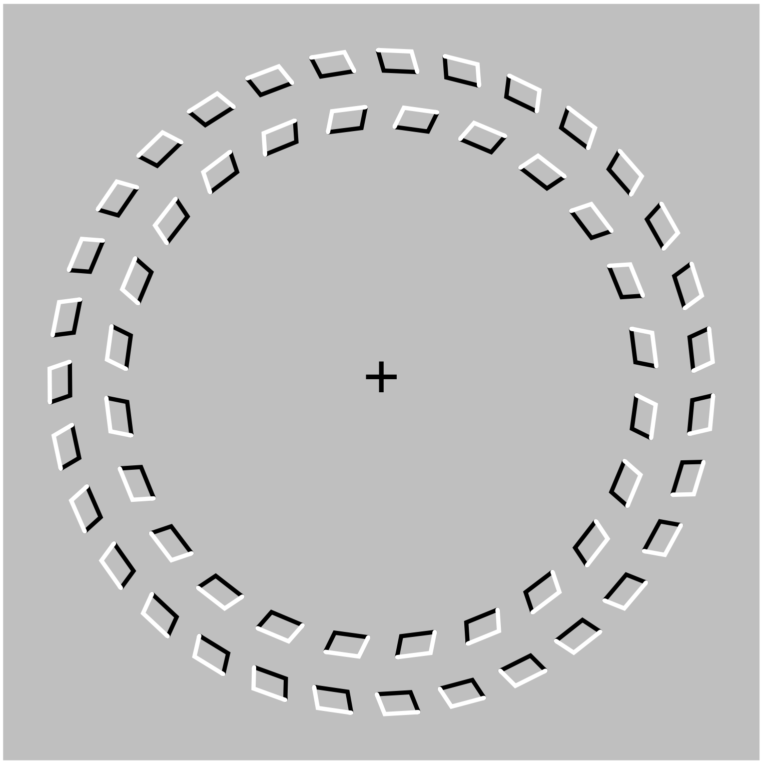
And, to finish, we redraw a picture of the Italian specialist of vision Baingio Pinna.[69] Note the usage of different kinds of indexes in the same loops and the change of the degrees parameter between the two loops.

```
\def\PstMyDiamond#1#2{{%
\psset{dimen=middle,unit=0.1,linewidth=2\pslinewidth,arrows=c-c}%
\psline(0,-3)(-2,#1)(0,3)
\psline[linecolor=white](0,-3)(2,#2)(0,3)}}

\pspicture(-5,-5)(5,5)
  \SpecialCoor
  \psframe*[linecolor=lightgray](-5,-5)(5,5)
  \psdot[dotstyle=B+,dotscale=3]
  \degrees[48]
  \multido{\iAngle=1+2,\nRotation=4.5+2.0}{24}{%
   \rput{\nRotation}(3.5;\iAngle){\PstMyDiamond{-1}{1}}}
```

---

[69] Fix 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.

```
  \degrees[62]
  \multido{\iAngle=1+2,\nRotation=-3.5+2.0}{31}{%
    \rput{\nRotation}(4.25;\iAngle){\PstMyDiamond{1}{-1}}}
\endpspicture
```



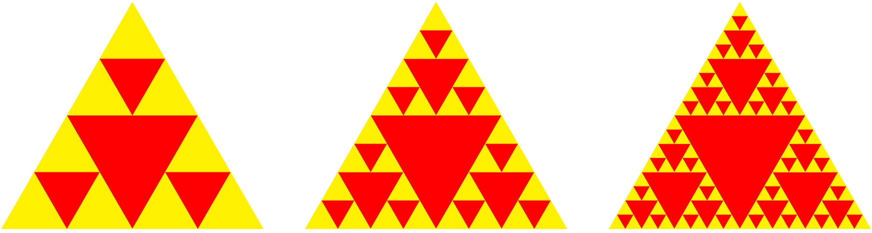
E Recursion

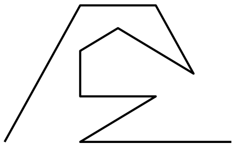
Recursion is a classical way of programming, which consist for a function to call itself with different parameters. Recursive algorithms are generally shortest than iterative ones, and are generally considered as elegant. Nevertheless, they are usually less efficient than iterative ones when executed 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.

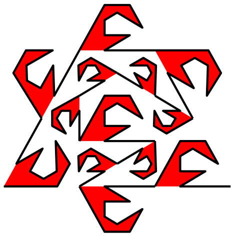
We give here some classical examples on fractals, for the well-known Sierpinski triangle (see for instance [49]) then for the 13 segments auto-similar curve shown by Benoit Mandelbrot in his book *The Fractal Geometry of Nature* [38].

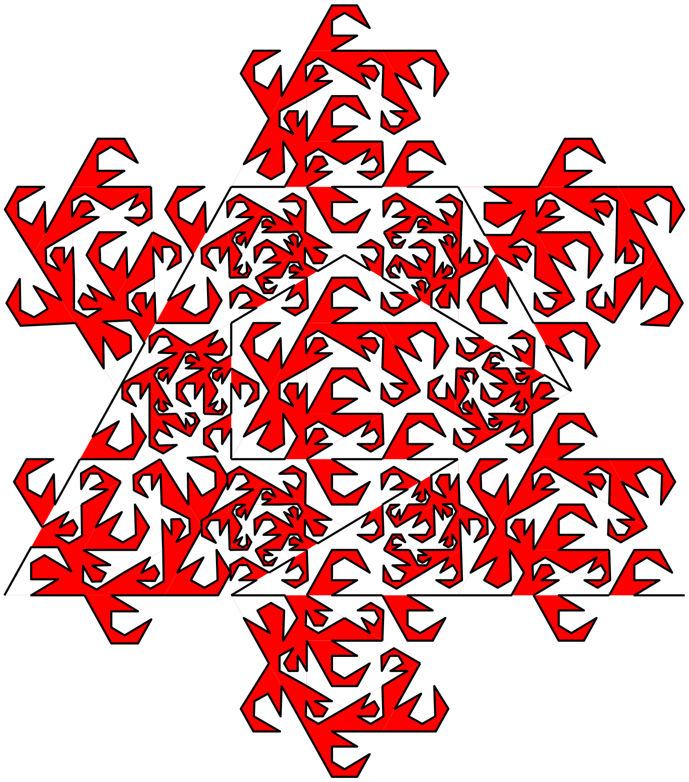
```
% From the source of LaTeX 2e (latex.ltx file)
\long\def\@firstoftwo#1#2{#1}
\long\def\@secondoftwo#1#2{#2}

```

```
% The recursion macro used (from David Carlisle)
\def\Recursion#1{%
#1\relax
   \expandafter\@firstoftwo
\else
   \expandafter\@secondoftwo
\fi}

\def\PstFractalBegin{}% Default value empty

% General definition of a fractal
\def\PstFractal{\def\pst@par{}\pst@object{PstFractal}}

\def\PstFractal@i#1{{%
\use@par% Assignment of local parameters
\PstFractalBegin% The first level can need special thing
\Recursion{\ifnum#1>\@ne}
           {\pst@cnth=#1\relax
            \advance\pst@cnth\m@ne
            \PstFractalRepeat{\pst@cnth}}
           {\PstFractalDefinition}}}

% Examples on classical fractals
% ------------------------------

% Sierpinski triangle
\def\PstSierpinskiTriangle#1{%
\def\PstFractalDefinition{%
   \pspolygon*[linecolor=\PstSierpinskiExternalColor](1;0)(1;1)(1;2)
   \rput{-2}(0,0){%
      \pspolygon*[linecolor=\PstSierpinskiInternalColor]
                    (0.5;0.5)(0.5;1.5)(0.5;2.5)}}%
\def\PstFractalBegin{\PstFractalDefinition}%
\def\PstFractalRepeat##1{%
   \rput(0.5;0){\PstFractal[unit=0.5]{##1}}
   \rput(0.5;1){\PstFractal[unit=0.5]{##1}}
   \rput(0.5;2){\PstFractal[unit=0.5]{##1}}}%
\pspicture(-0.866,-0.5)(0.866,1)
   \SpecialCoor
   \rput{90}(0,0){%
      \degrees[3]
      \PstFractal{#1}}
\endpspicture}

\def\PstMandelbrotCurve#1#2{%
\def\PstFractalDefinition{%
\pscustom[#2]{%
   \code{1 setlinejoin}%
   \moveto(0,0)
   \lineto(0.5,0.9)\lineto(1,1.8)\lineto(2,1.8)\lineto(2.5,0.9)
   \lineto(2,1.2)\lineto(1.5,1.5)\lineto(1,1.2)\lineto(1,0.6)
   \lineto(2,0.6)\lineto(1.5,0.3)\lineto(1,0)\lineto(2,0)\lineto(3,0)}}%
%
\def\PstFractalRepeat##1{{%
\psset{fillstyle=solid}%
\rput{60.95}(0,0){%
  \scalebox{1  -1}{\PstFractal[unit=0.3432,fillcolor=white]{##1}}}
\rput{60.95}(0.5,0.9){\PstFractal[unit=0.3432,fillcolor=red]{##1}}
\rput{0}(1,1.8){\PstFractal[unit=0.3333,fillcolor=red]{##1}}
\rput{-60.95}(2,1.8){\PstFractal[unit=0.3432,fillcolor=red]{##1}}
\rput{-210.96}(2.5,0.9){\PstFractal[unit=0.1944,fillcolor=red]{##1}}
\rput{-210.96}(2,1.2){%
  \scalebox{1  -1}{\PstFractal[unit=0.1944,fillcolor=white]{##1}}}
\rput{-149.04}(1.5,1.5){%
  \scalebox{1  -1}{\PstFractal[unit=0.1944,fillcolor=white]{##1}}}
\rput{-90}(1,1.2){%
  \scalebox{1  -1}{\PstFractal[unit=0.2,fillcolor=white]{##1}}}
\rput(1,0.6){\PstFractal[unit=0.3333,fillcolor=red]{##1}}
\rput{-149.04}(2,0.6){\PstFractal[unit=0.1944,fillcolor=red]{##1}}
\rput{-149.04}(1.5,0.3){%
  \scalebox{1  -1}{\PstFractal[unit=0.1944,fillcolor=white]{##1}}}
\rput(1,0){%
  \scalebox{1  -1}{\PstFractal[unit=0.3333,fillcolor=red]{##1}}}
\rput(2,0){\PstFractal[unit=0.3333,fillcolor=red]{##1}}}}%
%
\PstFractal{#1}}

\bgroup
  \def\PstSierpinskiInternalColor{red}%
  \def\PstSierpinskiExternalColor{yellow}%
  \psset{unit=2}%
  \PstSierpinskiTriangle{2}\hfill%
  \PstSierpinskiTriangle{3}\hfill%
  \PstSierpinskiTriangle{4}
\egroup

\PstMandelbrotCurve{1}{}
\hspace{2cm}
\PstMandelbrotCurve{2}{fillstyle=solid,fillcolor=red}

\bgroup
  \psset{unit=3}%
  \PstMandelbrotCurve{2}{linestyle=none,fillstyle=solid,fillcolor=red}
  \PstMandelbrotCurve{3}{}
\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 'fltpoint' [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 segments[71] which represent the fractal at it 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]

```
\pscustom{%
\moveto(0,0)
\lineto(1,0)
\lineto(1.5,0.866)
\lineto(2,0)
\lineto(3,0)
}
```

and the generated code for the recursion will be:

```
\rput{0}(0,0){\PstFractal[unit=0.33333]{#1}}
\rput{59.99927}(1,0){\PstFractal[unit=0.33333]{#1}}
\rput{-59.99927}(1.5,0.866){\PstFractal[unit=0.33333]{#1}}
\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

---

[70] Take 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.

[71] This 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]).

[72] This 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!

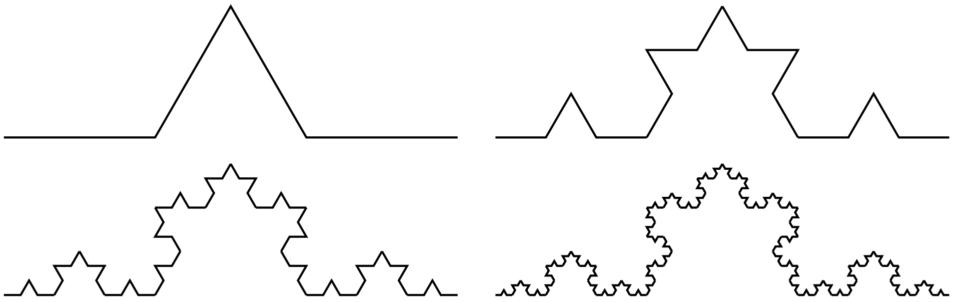
have the advantage to be able to reuse the generated codes for fractals already handled. This is not really more complicated to program, but the code is significantly longer, so we will not do it here.

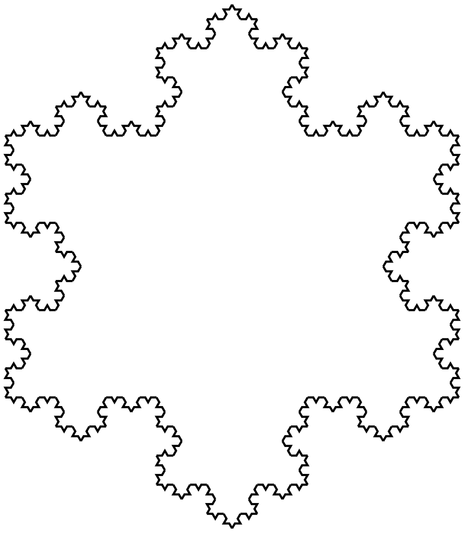
And as the computing of the rotation angle for all the repeated patterns require some trigonometric computations, we will use the ‘fp’ package [19] for that (see the macro \PstFractal@GenerateCommands@iii).

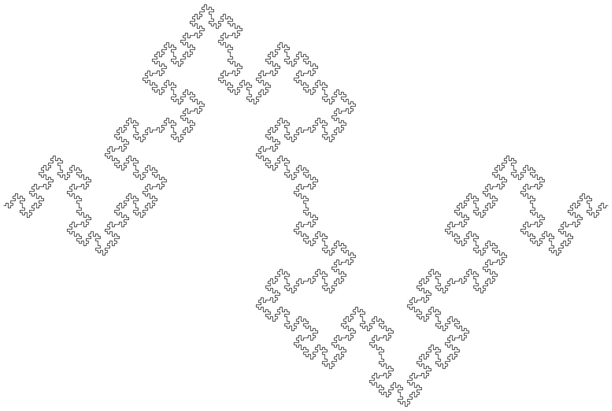
```
\def\@enamedef#1{\expandafter\edef\csname #1\endcsname}

% To compute hypotenuse
\def\FPhypotenuse#1#2#3{%
{\FPmul{\FPquada}{#2}{#2}
  \FPmul{\FPquadb}{#3}{#3}
  \FPadd{\FPsumquad}{\FPquada}{\FPquadb}
  \FProot{\FP@tmp}{\FPsumquad}{2}
  \global\let\FP@tmp\FP@tmp}%
\let#1\FP@tmp}

% General definition of a fractal
\def\PstFractal{\def\pst@par{}\pst@object{PstFractal}}

\def\PstFractal@i#1{{%
\use@par% Assignment of local parameters
\Recursion
    {\ifnum#1>\@ne}
    {\pst@cnth=#1\relax
      \advance\pst@cnth\m@ne
      \multido{\i=\@ne+\@ne}{20}{% No more than 20 segments expected
          \@nameuse{PstFractal@Repeat\i}{\the\pst@cnth}}}
    {\pscustom{%
          \multido{\i=\@ne+\@ne}{20}{\@nameuse{PstFractal@Definition\i}}}}}}

% Initialisation of internal macros for the fractal definition
\Multido{\i=1+1}{20}{%
\@namedef{PstFractal@Definition\i}{}%
\@namedef{PstFractal@Repeat\i}#1{}}

% Macro for automatic generation of auto-similar fractals
% by definition of the pattern only
\def\PstFractal@SelfSimilar{%
\def\pst@par{}%
\pst@object{PstFractal@SelfSimilar}}

\def\PstFractal@SelfSimilar@i#1#2{%
\use@par% Affectation of local parameters
\def\PstFractal@Deep{#1}%
\def\PstFractal@Scale{#2}%
\PstFractal@GenerateCommands}

\pst@cnta=\z@% Counter for the number of segments

```

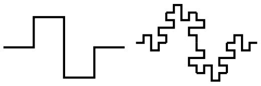
```
\def\PstFractal@GenerateCommands(#1){%
\advance\pst@cnta\@ne
\PstFractal@GenerateCommands@ii(#1)
\@ifnextchar({\PstFractal@GenerateCommands}
  {\PstFractal{\PstFractal@Deep}}}

\def\PstFractal@GenerateCommands@ii(#1,#2){%
\ifnum\pst@cnta>\@ne
  \PstFractal@GenerateCommands@iii
    (\PstFractal@PreviousX,\PstFractal@PreviousY,#1,#2)
\fi
\def\PstFractal@PreviousX{#1}%
\def\PstFractal@PreviousY{#2}}

\def\PstFractal@GenerateCommands@iii(#1,#2,#3,#4){%
\FPeval{\Xdiff}{abs(#3 - #1)}
\FPeval{\Ydiff}{abs(#4 - #2)}
\FPhypotenuse{\PstFractalHypothenuse}{\Xdiff}{\Ydiff}
% \FPeval easily generate overflows on next computations,
% so we rather use here the \FPupn variant
\FPupn{\PstFractalRotation}
  {\PstFractalHypothenuse{} \Xdiff{} / arccos 57.29578 *}
\ifdim#1\p@<#3\p@
  \ifdim#2\p@>#4\p@
    \FPneg{\PstFractalRotation}{\PstFractalRotation}
  \fi
\else
  \ifdim#2\p@<#4\p@
    \FPadd{\PstFractalRotation}{\PstFractalRotation}{180}
  \else
    \FPsub{\PstFractalRotation}{180}{\PstFractalRotation}
  \fi
  \FPneg{\PstFractalRotation}{\PstFractalRotation}
\fi
\FPround{\PstFractalRotation}{\PstFractalRotation}{5}
% Using \moveto and \lineto macros rather than \psline will optimize
% resulting file with a noticeable factor (around 2)!
\ifnum\pst@cnta=\tw@
  \@enamedef{PstFractal@Definition\the\pst@cnta}{%
    \noexpand\moveto(#1,#2)
    \noexpand\lineto(#3,#4)}%
\else
  \@enamedef{PstFractal@Definition\the\pst@cnta}{%
    \noexpand\lineto(#1,#2)
    \noexpand\lineto(#3,#4)}%
\fi
\@enamedef{PstFractal@Repeat\the\pst@cnta}##1{%
  \noexpand\rput{\PstFractalRotation}(#1,#2){%
    \noexpand\PstFractal[unit=\PstFractal@Scale]{##1}}}}

\def\PstVonKochCurve{%
\@ifnextchar[{\PstVonKochCurve@i}{\PstVonKochCurve@i[]}}
```

```
\def\PstVonKochCurve@i[#1]#2{%
\PstFractal@SelfSimilar[#1]{#2}{0.33333}(0,0)(1,0)(1.5,0.866)(2,0)(3,0)}

\def\PstVonKochSnake{%
\@ifnextchar[{\PstVonKochSnake@i}{\PstVonKochSnake@i[]}}

\def\PstVonKochSnake@i[#1]#2{%
\rput(0,0){\PstVonKochCurve[#1]{#2}}
\rput{240}(3,0){\PstVonKochCurve[#1]{#2}}
\scalebox{-1  1}{\rput{240}{\PstVonKochCurve[#1]{#2}}}}

\def\PstMinkowskiCurve{%
\@ifnextchar[{\PstMinkowskiCurve@i}{\PstMinkowskiCurve@i[]}}

\def\PstMinkowskiCurve@i[#1]#2{%
\PstFractal@SelfSimilar[#1]{#2}{0.25}%
    (0,0)(1,0)(1,1)(2,1)(2,0)(2,-1)(3,-1)(3,0)(4,0)}

\psset{unit=2}%

\pspicture(3,0.9)\PstVonKochCurve{1}\endpspicture
\hfill
\pspicture(3,0.9)\PstVonKochCurve{2}\endpspicture

\pspicture(3,0.9)\PstVonKochCurve{3}\endpspicture
\hfill
\pspicture(3,0.9)\PstVonKochCurve{4}\endpspicture

\pspicture(0,-2.6)(3,0.9)\PstVonKochSnake{4}\endpspicture

\pspicture(0,-0.2)(0.8,0.2)\PstMinkowskiCurve[unit=0.2]{1}\endpspicture
\hfill
\pspicture(0,-0.23)(0.8,0.23)\PstMinkowskiCurve[unit=0.2]{2}\endpspicture
\hfill
\pspicture(0,-1.3)(4,1.3)
    \PstMinkowskiCurve[linewidth=0.2\pslinewidth]{4}
\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 Zhuhan Jiang is very well adapted and powerful.[73] It allow to store and retrieve data in mono or bi-dimensional arrays. Here is an example which allow 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]

```
\def\PstPieChart#1#2#3#4{%
% #1 = Values, #2 = Labels, #3 = Styles, #4 = Starting angle
\pspicture(-2,-2)(2,2)
```

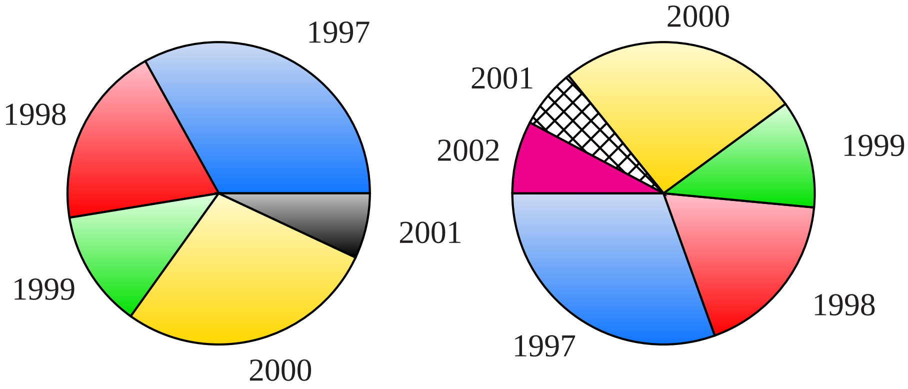
---

[73] Take care that it has an incompatibility with the ‘array’ LaTeX 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.

[74] The ‘pst-chrt’ contribution package [57], to draw various kinds of 2d and pseudo-3d business charts, implement such techniques.

```
  % To compute in \pst@dimd the sum of all the values
  \pst@dimd=\z@
  \Multido{\iNbValues=\z@+\@ne}{9999}{%
    \csname check#1\endcsname(\multidocount)%
    \ifemptydata
      \multidostop
    \else
      \advance\pst@dimd by \cachedata\p@
    \fi}
  \degrees[\pst@number{\pst@dimd}]
  % To compute the starting angle
  \pst@dimh=\pst@dimd
  \divide\pst@dimh by 360
  \multiply\pst@dimh by #4
  % Loop on slices
  \multido{}{\iNbValues}{%
    % Slice
    \pst@dimg=\pst@dimh
    \csname check#1\endcsname(\multidocount)%
    \advance\pst@dimh by \cachedata\p@
    \csname check#3\endcsname(\multidocount)%
    \pswedge[style=\cachedata]{2}
                {\pst@number{\pst@dimg}}{\pst@number{\pst@dimh}}
    % Label
    \advance\pst@dimg\pst@dimh
    \divide\pst@dimg\tw@
    \uput{2.2}[\pst@number{\pst@dimg}](0,0){%
      \large\csname #2\endcsname(\multidocount)}}
\endpspicture}

\newarray{\Values}
\newarray{\Labels}
\newarray{\Styles}

\readarray{Values}{32.7 & 19.3 & 12.4 & 27.6 & 6.9}
% Don’t leave blanks between labels and between styles!
\readarray{Labels}{1997&1998&1999&2000&2001}
\readarray{Styles}{StyleGradientA&StyleGradientB&%
  StyleGradientC&StyleGradientD&StyleGradientE}

\newpsstyle{}{}% For undefined styles
\newpsstyle{StyleGradientA}{%
  fillstyle=gradient,gradbegin=LightBlue,gradend=NavyBlue}
\newpsstyle{StyleGradientB}{%
  fillstyle=gradient,gradbegin=Pink,gradend=red}
\newpsstyle{StyleGradientC}{%
  fillstyle=gradient,gradbegin=PaleGreen,gradend=ForestGreen}
\newpsstyle{StyleGradientD}{%
  fillstyle=gradient,gradbegin=LemonChiffon,gradend=Gold}
\newpsstyle{StyleGradientE}{%
  fillstyle=gradient,gradbegin=lightgray,gradend=black}
\newpsstyle{StyleHatchA}{fillstyle=crosshatch}
\newpsstyle{StyleSolidA}{fillstyle=solid,fillcolor=magenta}
```

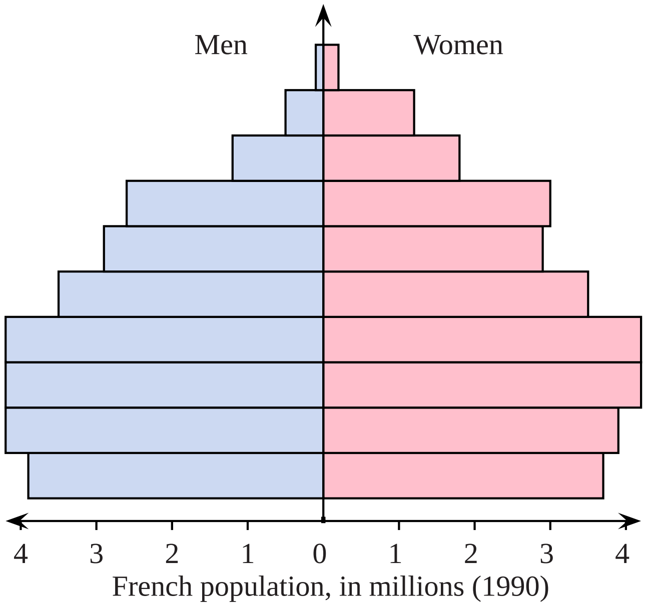
```

\psset{gradmidpoint=1}%

\PstPieChart{Values}{Labels}{Styles}{0}
\hspace{1.5cm}
\Styles(5)={StyleHatchA}
\Values(6)={8.3}\Labels(6)={2002}\Styles(6)={StyleSolidA}
\PstPieChart{Values}{Labels}{Styles}{180}
```



Then another example to draw a classical pyramid of ages in a generic way, automatically computing the size of the graphic and the axes.

```
\def\PstPyramidAges#1#2#3#4{{%
% #1 = Values for first set, #2 = Values for second set,
% #3 = Labels, #4 = Styles
\psset{dimen=middle}%
% To store in \pst@tempc the maximum value for the first set
% and in \pst@tempd the maximum value for the second set.
% This will allow to compute the size of the graphic
% and the correct lengths of the axes.
\pst@dimg=\z@
\pst@dimh=\z@
\Multido{\iNbValues=\z@+\@ne}{9999}{%
  \csname check#1\endcsname(\multidocount)%
  \ifemptydata
    \multidostop
  \else
    \ifdim\pst@dimg<\cachedata\p@
      \pst@dimg=\cachedata\p@
    \fi
    \csname check#2\endcsname(\multidocount)%
    \ifdim\pst@dimh<\cachedata\p@
      \pst@dimh=\cachedata\p@
    \fi
  \fi}
\edef\pst@tempc{\pst@number{\pst@dimg}}%
\edef\pst@tempd{\pst@number{\pst@dimh}}%
\pspicture(-\pst@tempc,-2.5)(\pst@tempd,\iNbValues.9)
  % The two styles used
```

```
   \csname check#4\endcsname(\@ne)%
   \let\PstStyleFirstSet\cachedata
   \csname check#4\endcsname(\tw@)%
   \let\PstStyleSecondSet\cachedata
   % Loop on values
   \multido{\iValue=\z@+\@ne}{\iNbValues}{%
      % First set
      \csname check#1\endcsname(\multidocount)%
      \psframe[style=\PstStyleFirstSet]
                (0,\iValue)(-\cachedata,\multidocount)
      % Second set
      \csname check#2\endcsname(\multidocount)%
      \psframe[style=\PstStyleSecondSet]
                (0,\iValue)(\cachedata,\multidocount)}
   % Axes
   \def\pshlabel##1{%
   \ifnum##1<\z@
      \pst@cnth=##1
      \pst@cnth=-\pst@cnth
      \the\pst@cnth
   \else
      ##1
   \fi}%
   \psaxes[tickstyle=bottom,ticks=x,labels=x,arrowscale=2]
            {<->}(0,-0.5)(-\pst@tempc,-0.5)(\pst@tempd,\iNbValues.9)
   % Labels and legend
   \rput[r](-1,\iNbValues){\csname #3\endcsname(\@ne)}
   \rput[l](1,\iNbValues){\csname #3\endcsname(\tw@)}
   \rput(0,-2){\csname #3\endcsname(\thr@@)}
\endpspicture}}

\newarray{\ValuesMen}
\newarray{\ValuesWomen}
\newarray{\Labels}
\newarray{\Styles}

\readarray{ValuesMen}{%
   3.9 & 4.2 & 4.2 & 4.2 & 3.5 & 2.9 & 2.6 & 1.2 & 0.5 & 0.1}
\readarray{ValuesWomen}{%
   3.7 & 3.9 & 4.2 & 4.2 & 3.5 & 2.9 & 3.0 & 1.8 & 1.2 & 0.2}

% Don’t leave blanks between labels and between styles!
\readarray{Labels}{Men&Women&%
   French population, in millions (1990)}
\readarray{Styles}{StyleSolidA&StyleSolidB}

\newpsstyle{StyleSolidA}{fillstyle=solid,fillcolor=LightBlue}
\newpsstyle{StyleSolidB}{fillstyle=solid,fillcolor=Pink}

\psset{yunit=0.6}%
\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.

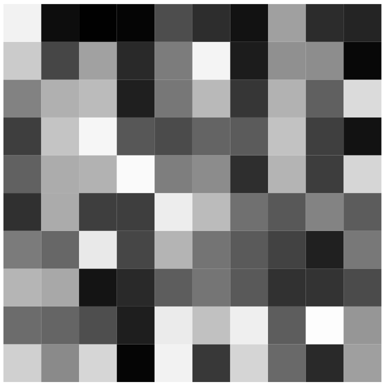
We give here examples of both usage, first to generate a patchwork of little squares of a random level of gray color. In the TeX version, we cannot use the **\multips** macro, as we need to generate at each iteration not only the code of a pure PSTricks graphic object, but also the code defining the new color. In the PostScript version, we must redefine the internal \pst@usecolor macro.

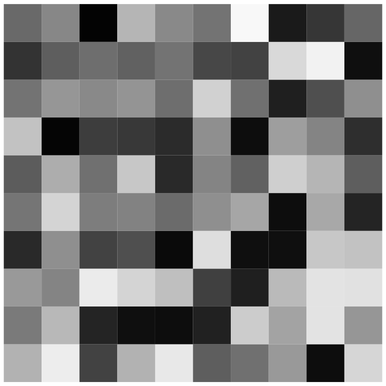
```
\multido{\nXA=0.0+0.5,\nXB=0.5+0.5}{10}{%
  \multido{\nYA=0.0+0.5,\nYB=0.5+0.5}{10}{%
```

---

[75] You 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.

```
    \setrandim{\pst@dimh}{0pt}{1pt}%
    \definecolor{MyGray}{gray}{\pst@number{\pst@dimh}}%
    \psframe*[linecolor=MyGray](\nXA,\nYA)(\nXB,\nYB)}}
\hfill
\multips(0,0.5){10}{%
  \multips(0.5,0){10}{%
    \def\pst@usecolor#1{rand 101 mod 100 div setgray }%
    \psframe*(0.5,0.5)}}
```

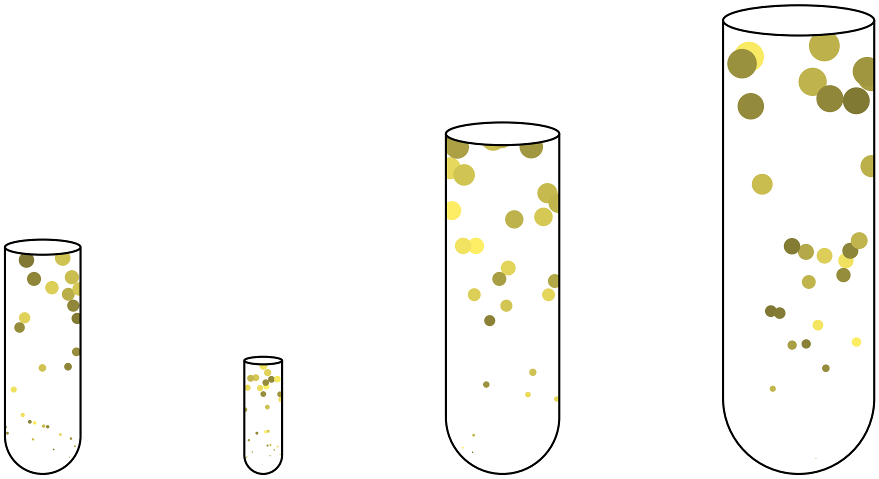




In the next example, we draw colored bubbles in a test tube. The position of the bubbles is randomly computed, as their sizes, which may also depend of the value of the current unit, and they also grow according to the vertical position inside the tube. The color of the bubbles is also set randomly, using the Brightness component of the HSB color model.

```
\def\PstTestTube{%
\psline[linearc=0.5](-0.5,3)(-0.5,0)(0.5,0)(0.5,3)
\psellipse[fillstyle=solid,fillcolor=white](0,3)(0.5,0.1)}

\def\PstTestTubeBubbles{%
\pspicture(-0.5,0)(0.5,3.1)
\psclip{\pscustom[linestyle=none]{\PstTestTube}}
  \multido{}{30}{%
    \setrandim{\pst@dimc}{-0.5pt}{0.5pt}% Horizontal coordinate
    \setrandim{\pst@dimd}{0pt}{3pt}%      Vertical coordinate
    \setrandim{\pst@dimg}{0.5pt}{1pt}%    Color brightness
    \definecolor{MyColor}{hsb}{0.15,0.6,\pointless\pst@dimg}%
    % The size increase proportionally to the vertical position
    \pst@dimh=\pst@dimd
    % And depend of the unit
    \pst@dimg=0.02\psunit
    \pst@dimh=\pointless\pst@dimg\pst@dimh
    \psdot[dotscale=\pointless\pst@dimh,linecolor=MyColor]
      (! \pst@number{\pst@dimc} \pst@number{\pst@dimd})}
\endpsclip
\PstTestTube
\endpspicture}

```

```
\PstTestTubeBubbles\hfill
{\psset{unit=0.5}\PstTestTubeBubbles}\hfill
{\psset{unit=1.5}\PstTestTubeBubbles}\hfill
{\psset{unit=2}\PstTestTubeBubbles}
```



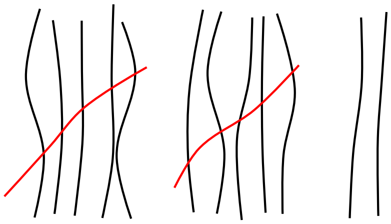
Now, a macro to draw fuzzy lines, to mimic hand made bars.[76] It allow to draw sticks to show quantities.

```
\def\PstLineFuzzy[#1](#2,#3)(#4,#5)(#6,#7)(#8,#9){%
\pscurve[#1](! #2 rand 101 mod 1000 div sub
                #3 rand 101 mod 1000 div sub)
             (! #4 rand 101 mod 1000 div sub
                #5 rand 101 mod 1000 div sub)
             (! #6 rand 101 mod 1000 div sub
                #7 rand 101 mod 1000 div sub)
             (! #8 rand 101 mod 1000 div sub
                #9 rand 101 mod 1000 div sub)}

\def\PstSticks#1{%
\multido{\iStick=0+1,\nXA=0.1+0.1,\nXB=-0.5+0.1,
          \nXC=-0.35+0.10,\nXD=-0.15+0.10}{#1}{%
  \pst@cnta=\iStick
  \pst@cnth=\iStick
  \divide\pst@cnth by 5
  \multiply\pst@cnth by 5
  \ifnum\iStick>0\relax
    \ifnum\pst@cnta=\pst@cnth
      \PstLineFuzzy[linecolor=red]%
                    (\nXB,0.2)(\nXC,0.4)(\nXD,0.6)(\nXA,0.8)
      \hbox to 0.2\psxunit{}
    \fi
  \fi
  \PstLineFuzzy[](\nXA,0.1)(\nXA,0.4)(\nXA,0.7)(\nXA,1)}}

```

---

[76] The initial idea come from MetaFun [40] from Hans Hagen.

```
\pspicture(-0.1,0)(3.6,3)
  \psset{unit=3}%
  \PstSticks{12}
\endpspicture

\large
\psset{xunit=0.8,yunit=0.3}%
\begin{tabular}{|l|r|p{5.5cm}|}
  \hline
  Linux        & 27 & \PstSticks{27} \\ \hline
  Mac OS X     & 14 & \PstSticks{14} \\ \hline
  Windows XP & 43 & \PstSticks{43} \\ \hline
\end{tabular}
```



Linux	27	
Mac OS X	14	
Windows XP	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]

```
\newarray{\PstMosaic@Indexes}

\def\PstMosaic(#1,#2)(#3,#4)#5#6{{%
% #1,#2 = X and Y dimensions (without unit)
% #3,#4 = horizontal and vertical number of shifts
% #5 = PSTricks graphic code or external image
% #6 = PSTricks parameters for the borders
\expandarrayelementtrue
\dataheight=#3\relax
% Initialization of an array of indexes
\multido{\iX=\@ne+\@ne}{#3}{%
```

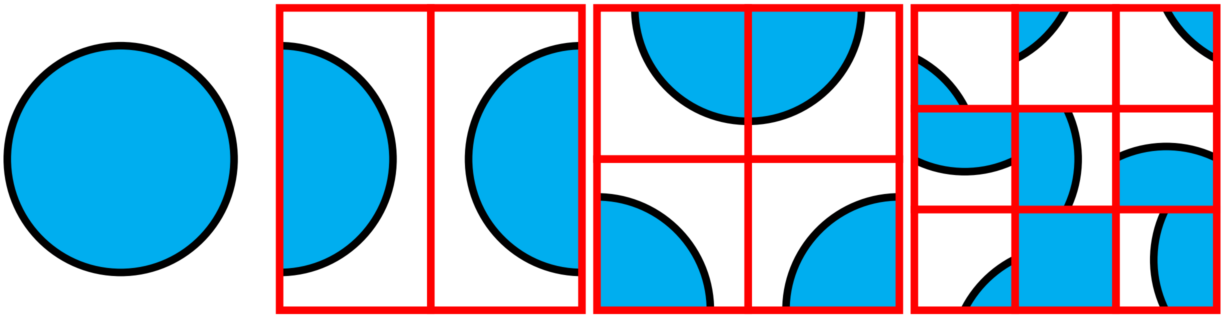
---

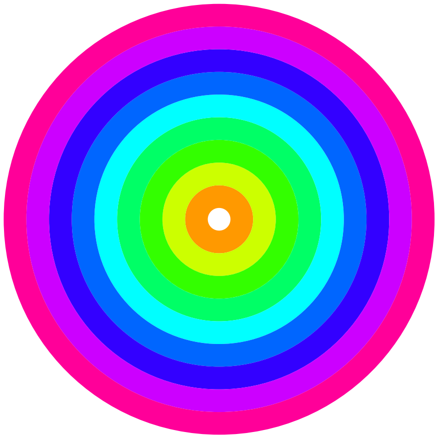
[77] The initial idea come from MetaFun [40] from Hans Hagen.

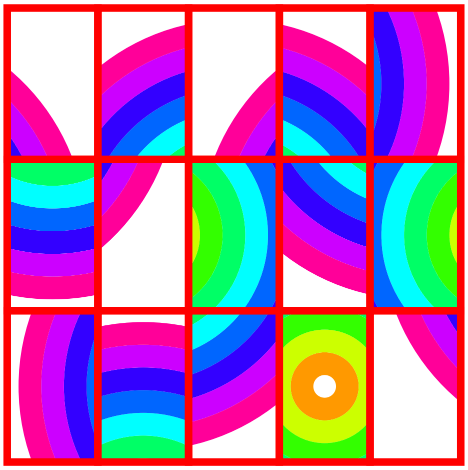
[78] Take care that in this last case the image is loaded the number of times corresponding to the number of cells, which can generate a huge resulting file if the compiler used is not able to load the file only one time, or if you do not use a special EPS file with a core part which will be loaded only one time (see the paragraph 15 (in it version 2.0) in the document of Keith Reickdahl about the insertion of EPS images with LaTeX [75]).

```
  \multido{\iY=\@ne+\@ne}{#4}{%
    \PstMosaic@Indexes(\iX,\iY)={(\iX,\iY)}}}
% Shuffling of this array of indexes (50 times must be enough!)
\PstMosaic@IndexesShuffle(#3,#4){50}%
% The mosaic
\pst@dimg=#1\p@
\divide\pst@dimg by #3
\pst@dimh=#2\p@
\divide\pst@dimh by #4
\pst@dimtonum{\pst@dimg}{\pst@tempi}%
\pst@dimtonum{\pst@dimh}{\pst@tempj}%
\pspicture(#1,#2)
  \multido{\iX=\@ne+\@ne}{#3}{%
    \multido{\iY=\@ne+\@ne}{#4}{%
      \checkPstMosaic@Indexes(\iX,\iY)%
      \expandafter\PstMosaic@GetIndexes\cachedata
      \PstMosaic@Tile(\pst@tempi,\pst@tempj)%
        (\pst@tempg,\pst@temph)(\iX,\iY){#5}{#6}}}
\endpspicture}}

\def\PstMosaic@IndexesShuffle(#1,#2)#3{%
\multido{\iShuffle=\@ne+\@ne}{#3}{%
  \setrannum{\pst@cnta}{\@ne}{#1}%
  \setrannum{\pst@cntb}{\@ne}{#2}%
  \setrannum{\pst@cntc}{\@ne}{#1}%
  \setrannum{\pst@cntd}{\@ne}{#2}%
  % Exchange of \PstMosaic@Indexes(\pst@cnta,\pst@cntb)
  % and \PstMosaic@Indexes(\pst@cntc,\pst@cntd)
  \checkPstMosaic@Indexes(\the\pst@cnta,\the\pst@cntb)%
  \let\pst@tempg\cachedata
  \checkPstMosaic@Indexes(\the\pst@cntc,\the\pst@cntd)%
  \PstMosaic@Indexes(\the\pst@cnta,\the\pst@cntb)={\cachedata}%
  \PstMosaic@Indexes(\the\pst@cntc,\the\pst@cntd)={\pst@tempg}}}

\def\PstMosaic@GetIndexes(#1,#2){%
\edef\pst@tempg{#1}%
\edef\pst@temph{#2}}

\def\PstMosaic@Tile(#1,#2)(#3,#4)(#5,#6)#7#8{%
\PstMosaic@Position{#1}{#3}{\pst@tempa}%
\PstMosaic@Position{#2}{#4}{\pst@tempb}%
\PstMosaic@Position{#1}{#5}{\pst@tempc}%
\PstMosaic@Position{#2}{#6}{\pst@tempd}%
\rput(\pst@tempc,\pst@tempd){%
  \psclip{\psframe[linestyle=none](#1,#2)}
    \rput(-\pst@tempa,-\pst@tempb){#7}%
  \endpsclip
  \psframe[dimen=middle,#8](#1,#2)}}

\def\PstMosaic@Position#1#2#3{%
\pst@dimh=#1\p@
\pst@cnth=#2
\advance\pst@cnth\m@ne
```

```
\multiply\pst@dimh\pst@cnth
\pst@dimtonum{\pst@dimh}{#3}}

% Examples
% ————————

\def\PstCircle{%
\pscircle[linewidth=0.1,fillstyle=solid,fillcolor=cyan](2,2){1.5}}

\pspicture(4,4)
  \PstCircle
\endpspicture
\hfill
\PstMosaic(4,4)(2,1){\PstCircle}{linecolor=red,linewidth=0.1}
\hfill
\PstMosaic(4,4)(2,2){\PstCircle}{linecolor=red,linewidth=0.1}
\hfill
\PstMosaic(4,4)(3,3){\PstCircle}{linecolor=red,linewidth=0.1}

\def\PstCircleHue{{%
\psset{unit=3}%
\multido{\nHue=0.0+0.1}{10}{%
  \definecolor{MyColor}{hsb}{\nHue,1,1}%
  \pscircle[linewidth=0.1,linecolor=MyColor](1,1){\nHue}}}}

\pspicture(6,6)
  \PstCircleHue
\endpspicture
\hfill
\PstMosaic(6,6)(5,3){\PstCircleHue}{linecolor=red,linewidth=0.1}

\includegraphics{lbrooks}% Louise Brooks (1906-1985)
\hfill
\PstMosaic(5.88,6.84)(3,2){\includegraphics[bb=0 0 1 1]{lbrooks}}
         {linestyle=dashed,linewidth=0.1}
```







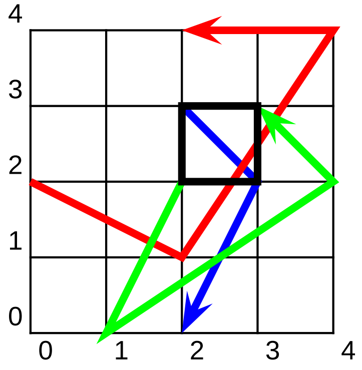
I Coordinates

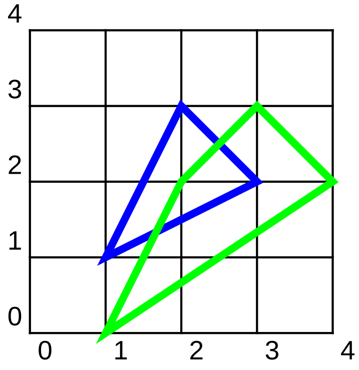
With it *special* coordinates (see Section 54), PSTricks offer powerful capabilities to handle coordinates. Nevertheless, one more thing which can be useful is to be able to handle relative coordinates, both for Cartesian and polar ones, and also for self defined lengths.

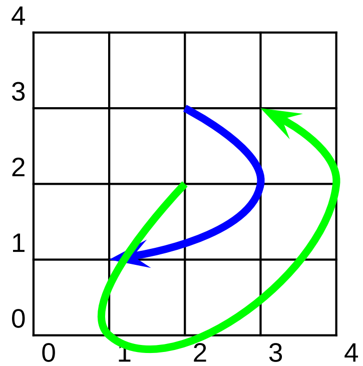
I.1 Relative coordinates

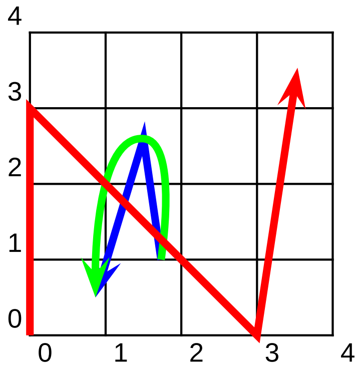
```
% D.G. - 1994/1997

% Minimal implementation of relative cartesian and polar
% coordinates
% (n:m) add n units in X direction and m in Y direction
% (n!m) add n units in X direction and m degrees of angle

% New functions
\def\psaddtoxlength#1#2{%
\let\@psunit\psxunit
\afterassignment\pstunit@off
\advance#1 #2\@psunit}

\def\psaddtoylength#1#2{%
\let\@psunit\psyunit
\afterassignment\pstunit@off
\advance#1 #2\@psunit}

\def\psaddtoangle#1{%
\edef\pst@angle{\pst@angle #1 \pst@angleunit add }}

\def\PstRelativeCartesian@coor#1#2{%
\psaddtoxlength{\pst@dimg}{#1}%
\psaddtoylength{\pst@dimh}{#2}%
\edef\pst@coor{\pst@number\pst@dimg \pst@number\pst@dimh}}

\def\PstRelativePolar@coor#1#2{%
\psaddtoxlength{\pst@dimg}{#1}%
\psaddtoangle{#2}%
\edef\pst@coor{\pst@number\pst@dimg \pst@angle \tx@PtoC}}

\def\special@@@coor#1;#2;#3\@nil{%
\ifx#3;\relax
  \polar@coor{#1}{#2}%
\else
  \special@@@@coor#1::\@nil
\fi}

\def\special@@@@coor#1:#2:#3\@nil{%
\ifx#3:\relax
  \PstRelativeCartesian@coor{#1}{#2}%
\else
  \special@@@@@coor#1!!\@nil
\fi}

\def\special@@@@@coor#1!#2!#3\@nil{%
\ifx#3!\relax
  \PstRelativePolar@coor{#1}{#2}%
\else
  \cartesian@coor#1,\relax,\@nil
\fi}
```

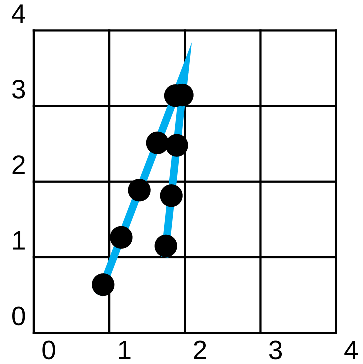
```
\psset{subgriddiv=0,linewidth=0.1,arrowscale=1.5}

\pspicture(4,4)\psgrid
   \psline[linecolor=blue]{->}(2,3)(3,2)(2,0)
   \psline[linecolor=red]{->}(0,2)(2:-1)(4,4)(-2:0)
   \psline[linecolor=green]{->}(2,2)(-1:-2)(3:2)(-1:1)
   \psframe(2,2)(1:1)
\endpspicture
\hfill
\pspicture(4,4)\psgrid
   \psset{xunit=2}%
   \pspolygon[linecolor=blue](1,3)(1.5,2)(0.5,1)
   \pspolygon[linecolor=green](1,2)(-0.5:-2)(1.5:2)(-0.5:1)
\endpspicture

\pspicture(4,4)\psgrid
   \pscurve[linecolor=blue]{->}(2,3)(3,2)(1,1)
   \pscurve[linecolor=green]{->}(2,2)(-1:-2)(3:2)(-1:1)
\endpspicture
\hfill
\pspicture(4,4)\psgrid
   \psline[linecolor=blue]{->}(2;30)(3;60)(1;30)
   \pscurve[linecolor=green]{->}(2;30)(1!30)(-2!-30)
   \psline[linecolor=red]{->}(0;0)(3!90)(0!-90)(2!45)
\endpspicture

\pspicture(4,4)\psgrid
   \psline[linecolor=cyan](2;30)(4;60)(1;30)
   \psline[linestyle=dotted,linewidth=0.3,dotsep=0.4](2;30)(2!30)(-3!-30)
\endpspicture
\hfill
\pspicture(4,4)\psgrid
   \degrees[4]
   \psline[linecolor=cyan](3;0)(4;1)(0;2)
   \psline[linestyle=dotted,linewidth=0.3,dotsep=0.4](3;0)(1!1)(-4!1)
\endpspicture
```

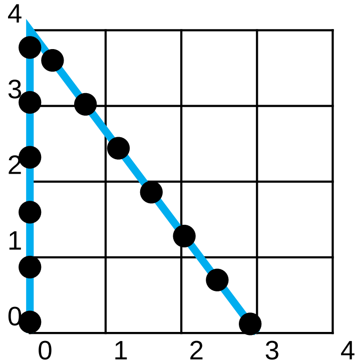










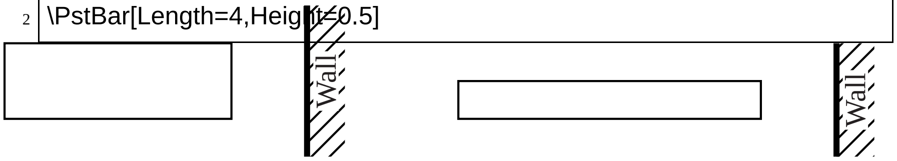


Another wish could be to define relative values of existing real or lengths parameters. This can be done using a macro like the \Pst@SetAbsOrRelDim@i one below, that we will illustrate on a modified version of the \PstBar macro built in the chapter XII, which define it parameters using the interface of the 'pst-key' package.

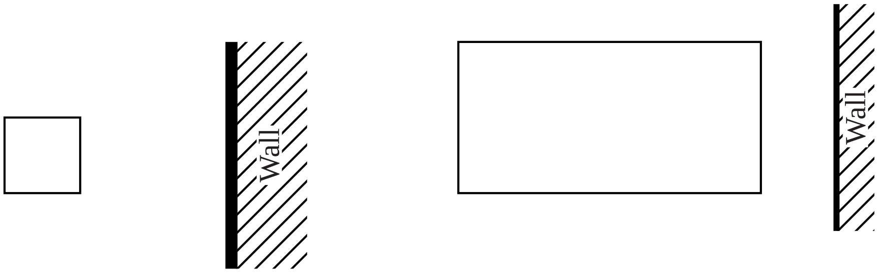
```
\define@key{psset}{Length}{%
\Pst@SetAbsOrRelDim@i{\PstBar@Length}#1\@nil}

\define@key{psset}{Height}{%
\Pst@SetAbsOrRelDim@i{\PstBar@Height}#1\@nil}

\def\Pst@SetAbsOrRelDim@i#1{%
\@ifnextchar+{\Pst@SetAbsOrRelDim@ii{#1}}
             {\Pst@SetAbsOrRelDim@iii{#1}}}

\def\Pst@SetAbsOrRelDim@ii#1+#2\@nil{%
\pssetlength{\pst@dimh}{#1}%
\psaddtolength{\pst@dimh}{#2}%
\edef#1{\the\pst@dimh}}

\def\Pst@SetAbsOrRelDim@iii#1#2\@nil{\edef#1{#2}}

\setkeys{psset}{Length=3,Height=1}% Default values

\def\PstBar{\@ifnextchar[{\PstBar@i}{\PstBar@i[]}}
```

```

\def\PstBar@i[#1]{{%
\setkeys{psset}{framesep=1pt,#1}%
\psframe(\PstBar@Length,\PstBar@Height)
\pssetlength{\pst@dima}{\PstBar@Length}%
\psaddtolength{\pst@dima}{\@ne}%
\pssetlength{\pst@dimc}{\PstBar@Height}%
\psaddtolength{\pst@dimc}{0.5}%
\psline[linewidth=0.08](\pst@dima,-0.5)(\pst@dima,\pst@dimc)
\pst@dimb=\pst@dima
\psaddtolength{\pst@dimb}{0.5}%
\psframe[linestyle=none,fillstyle=hlines]
            (\pst@dima,-0.5)(\pst@dimb,\pst@dimc)
\psaddtolength{\pst@dima}{0.25}%
\pssetlength{\pst@dimc}{\PstBar@Height}%
\divide\pst@dimc\tw@
\rput*{90}(\pst@dima,\pst@dimc){Wall}}}
```

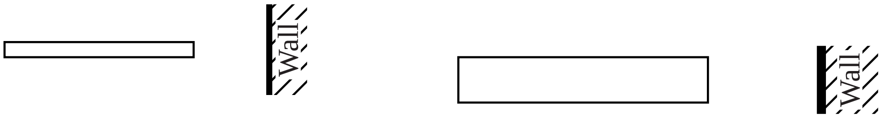
```
\rput(-6,0){\PstBar}
\PstBar[Length=4,Height=0.5]
```



```
\rput(-6,0){\PstBar[unit=2,Length=1cm,Height=1cm]}
\PstBar[Length=+1cm,Height=+1cm]
```



```
\rput(-6,0){%
   \setkeys{psset}{Length=4}%
   \PstBar[Length=+-15mm,Height=+-0.8cm]}
\PstBar[unit=1.5,Length=+-0.8,Height=+-1.4]
```



J Nodes

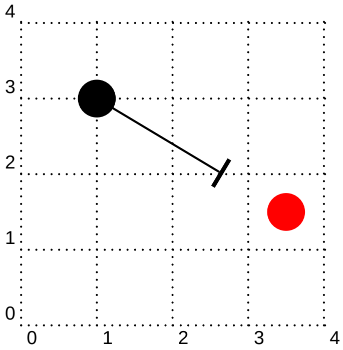
Nodes are a very powerful way to handle coordinates.[79]

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.



```
\SpecialCoor
\Cnode*(1,3){A}
\Cnode*[linecolor=red](3.5,1.5){B}
\psline[arrowscale=2]{-|}(A)
   (! tx@NodeDict begin% 2/3 of the AB segment
         /N@A load GetCenter % Center of node A
         /yA ED
         /xA ED
         /N@B load GetCenter % Center of node B
         /yB ED
         /xB ED
         xB xA sub 0.66667 mul xA add \pst@number\psxunit div
         yB yA sub 0.66667 mul yA add \pst@number\psyunit div
      end)
```

---

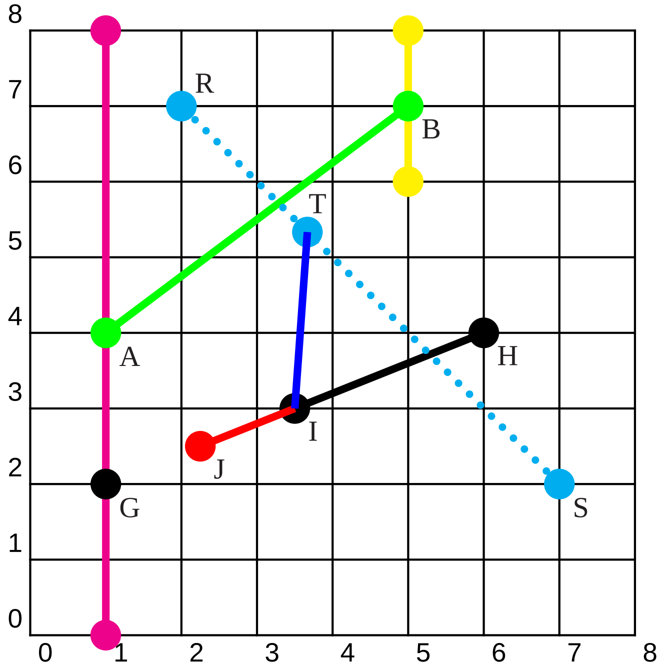
[79] The contribution packages ‘pst-circ’ [58] from Christophe Jorssen and Herbert Voss, for electric circuits, and ‘pst-eucl’ [59] from Dominique Rodriguez, for Euclidean geometry drawings, are good illustrations of which high level generic functions they allow to built.

[80] This is because there is no possibility to sent back informations from the PostScript level to the TEX one during the compilation (the VTEX 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 got, with the conjunction of the **\pnode** and **\ncput** macros, together with the tpos parameter.

```
\SpecialCoor
\psset{linewidth=0.1,radius=0.5}%
\pcline[linecolor=magenta](1,0)(1,8)
\ncput{\pnode{A}}% A is the middle of (1,0) and (1,8) -> (1,4)
\pcline[linecolor=yellow](5,6)(5,8)
\ncput{\pnode{B}}% B is the middle of (5,6) and (5,8) -> (5,7)
\psline[linecolor=green](A)(B)
%
\pnode(1,2){G}
\pnode(6,4){H}
\ncline[linestyle=none]{G}{H}
% Join the middle of the segment joining G and H, with H
\ncput{\psline(H)}
%
% I is the middle of the segment joining G and H
\ncput{\pnode{I}}
% J is the middle of the segment joining G and I
\ncput[npos=0.25]{\pnode{J}}
\psline[linecolor=red](I)(J)
%
\psset{linecolor=cyan}%
\pnode(2,7){R}
\pnode(7,2){S}
\ncline[linestyle=dotted]{R}{S}
% T is the third of the segment joining R and S
\ncput[npos=0.3333]{\pnode{T}}
\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 LATEX 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 LATEX'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 LATEX \parbox or minipage. LATEX 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 LATEX LR-box commands.

With LATEX 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

```
\pslongbox{MyFrame}{\psframebox}
```

you can write[82]

```
\MyFrame  <stuff>\endMyFrame
```

instead of

```
\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:

---

[81] This is a token register.

[82] LATEX 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 LaTeX:

```
\pslongbox{MyFrame}{\psframebox}
\newenvironment{fminipage}%
  {\MyFrame\begin{minipage}}%
  {\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 TeX 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 LaTeX example to see the problem:

```
\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:

```
\pslongbox{MyFrame}{\psframebox}
\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:

```
\psverbboxtrue
\psframebox{\verb+\foo{bar}+}
```

---

[83] Incidentally, 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 section is to recheck.

L.1 How to inactivate the PSTricks macros

To suppress the PostScript code generate by PSTricks macros, put the command

\PSTricksOff

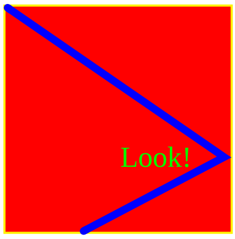
at the beginning of your document. You should then be able to print or preview drafts of your document (minus the PostScript, and perhaps pretty strange looking) with any dvi driver.

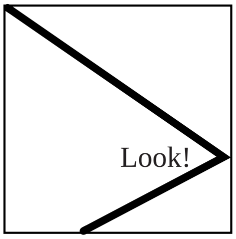
L.2 Monochrome version of all or some graphics

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 TeX or LaTeX, use the monochrome parameter of the ‘pstcol’ package (it will just pass this option to the ‘color’ one).

```
\documentclass[...]{...}

\usepackage[monochrome]{pstcol}
% ...

\makeatletter
\def\PSTricksMonochrome{%
\def\psset@bordercolor##1{\pst@getcolor{white}{\psbordercolor}}%
\def\psset@doublecolor##1{\pst@getcolor{white}{\psdoublecolor}}%
\def\psset@shadowcolor##1{\pst@getcolor{darkgray}{\psshadowcolor}}%
\def\psset@linecolor##1{\pst@getcolor{black}{\pslinecolor}}%
\def\psset@fillcolor##1{\pst@getcolor{white}{\psfillcolor}}%
\def\psset@gridcolor##1{\pst@getcolor{black}{\psgridcolor}}%
\def\psset@gridlabelcolor##1{\pst@getcolor{black}{\psgridlabelcolor}}%
\def\psset@hatchcolor##1{\pst@getcolor{black}{\pshatchcolor}}%
\def\psset@subgridcolor##1{\pst@getcolor{gray}{\pssubgridcolor}}}
\makeatother

%\PSTricksMonochrome% To force it on the whole document
```

```

\begin{document}

\begin{pspicture}(3,3)
  \psframe[linecolor=yellow,fillstyle=solid,fillcolor=red](3,3)
  \psline[linecolor=blue,linewidth=0.1]{cc-cc}(0,3)(2.9,1)(1,0)
  \rput(2,1){\textcolor{green}{Look!}}
\end{pspicture}
\hfill
\begin{pspicture}(3,3)
  \PSTricksMonochrome% To force locally monochrome rendering
  \psframe[linecolor=yellow,fillstyle=solid,fillcolor=red](3,3)
  \psline[linecolor=blue,linewidth=0.1]{cc-cc}(0,3)(2.9,1)(1,0)
  \rput(2,1){\textcolor{green}{Look!}}
\end{pspicture}

\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

```
\pstverb{106 45 {dup mul exch dup mul add 1.0 exch sub}
         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

```
\SpecialCoor
\def\mydata{23 43}
\psline(!47 \mydata add)
\psline(!47 \mydata\ add)
```

```
\psline(!47 \mydata~add)
\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.,

```
\def\mydata{23 43}
\edef\temp{47 \mydata add}
\show\temp
\edef\temp{47 \mydata\ add}
\show\temp
\edef\temp{47 \mydata~add}
\show\temp
\edef\temp{47 \mydata{} add}
\show\temp
```

TEX expands the code, assigns its value to `\temp`, and then displays the value of `\temp` on your console. Hit *return* to procede. You fill find that the four samples expand, respectively, to:

```
47 23 43add
47 23 43\ add
47 23 43\penalty \@M \ add
47 23 43{} add
```

All you really wanted was a space between the `43` and `add`. The command `\space` will do the trick:

```
\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 you 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.,

```
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 LATEX, this is logical to rather use the standard ConTEXt syntax to define and use colors. For that, a minor adaptation of the \dodefinecolor macro is needed. We also add the support of the HSB color model with the ConTEXt syntax, not currently supported by this TEX format.

```
% D.G. - April 2003

% To allow to define colors in Gray, RGB, CMYK and HSB models,
% using the ConTeXt syntax.

\def\dodefinecolor[#1][#2]{%
\doifassignmentelse{#2}{\dodefinecolori[#1][#2]#2\relax}{}}

\def\dodefinecolori[#1][#2]#3#4\relax{%
\edef\tempa{#3}%
\edef\tempb{s}%
\ifx\tempa\tempb% Gray model
   \getparameters[pstricks][s=0,#2]%
   \expanded{\newgray{#1}{\pstricksg}}%
\else
   \edef\tempb{r}%
   \ifx\tempa\tempb% RGB model
      \getparameters[pstricks][r=0,g=0,b=0,#2]%
      \expanded{\newrgbcolor{#1}{{\pstricksr} {\pstricksg} {\pstricksb}}}%
   \else
      \edef\tempb{c}%
      \ifx\tempa\tempb% CMYK model
         \getparameters[pstricks][c=0,m=0,y=0,k=0,#2]%
         \expanded{\newcmykcolor{#1}{{\pstricksc} {\pstricksm}
                                                   {\pstricksy} {\pstricksk}}}%
      \else
         \edef\tempb{H}%
         \ifx\tempa\tempb% HSB model
            \getparameters[pstricks][H=0,S=0,B=0,#2]%
            \expanded{\newhsbcolor{#1}{{\pstricksH} {\pstricksS} {\pstricksB}}}%
         \fi
      \fi
   \fi
\fi}

% Support of the HSB model
% ----------------------------
```

```

\def\dododefinecolor#1#2#3#4[#5][#6]% #2==set(g)value #3==set[e|x]value
  {#1\addtocommalist{#5}\colorlist
   \doifassignmentelse{#6}
     {\@@resetcolorparameters
      \getparameters[\??cl @@][#6]%
      \doifelse{\@@cl@@r\@@cl@@g\@@cl@@b}
             {\@@cl@@z\@@cl@@z\@@cl@@z}
        {\doifelse{\@@cl@@c\@@cl@@m\@@cl@@y\@@cl@@k}
               {\@@cl@@z\@@cl@@z\@@cl@@z\@@cl@@z}
% D.G. modification begin - May. 26, 2003
           {\doifelse{\@@cl@@H\@@cl@@S\@@cl@@B}
                  {\@@cl@@z\@@cl@@z\@@cl@@z}
% D.G. modification end
              {\doifelse\@@cl@@s\@@cl@@z
                 {\showmessage\m!colors8{{[#6]},#5}%
                  #3{\??cr#5}{\colorZpattern}}
                 {#3{\??cr#5}{\colorSpattern}}}
% D.G. modification begin - May. 26, 2003
              {#3{\??cr#5}{\colorHpattern}}}
% D.G. modification end
           {#3{\??cr#5}{\colorCpattern}}}
        {#3{\??cr#5}{\colorRpattern}}}
     {\doifdefinedelse{\??cr#6}
        {\doifelse{#5}{#6}
           {% this way we can freeze
            % \definecolor[somecolor][somecolor]
            % and still prevent cyclic definitions
            \iffreezecolors#3{\??cr#5}{\getvalue{\??cr#6}}\fi}
           {\iffreezecolors\@EA#3\else\@EA#2\fi
              {\??cr#5}{\getvalue{\??cr#6}}}}
        {\showmessage\m!colors3{#5}}}%
   \ifcase#4\or
     \unexpanded#2{#5}{\switchtocolor[#5]}% \unexpanded toegevoegd
   \fi}

\def\colorHpattern{%
0H:\@@cl@@H:\@@cl@@S:\@@cl@@B:\@@cl@@A:\@@cl@@t}
```

```
\starttext

% New colors (in ConTeXt syntax)
\definecolor[ForestGreen][r=0.13,g=0.55,b=0.13]
\definecolor[LemonChiffon][r=1,g=0.98,b=0.8]
\definecolor[LightBlue][r=0.8,g=0.85,b=0.95]
\definecolor[LightGray][s=0.92]
\definecolor[LightOrange][c=0,m=0.2,y=0.4,k=0]

\pspicture(-1.5,-1.5)(4.7,1.5)
  \psset{fillstyle=solid}%
  \pscircle[fillcolor=yellow]{1.5}
```

```
    \definecolor[LightBlue][r=0.8,g=0.85,b=0.95]
    \definecolor[LightOrange][c=0,m=0.2,y=0.4,k=0]
    \pscircle[linecolor=LightBlue,fillcolor=LightOrange]
             (3.2,0){1.5}
\endpspicture

\pspicture(-3,-3)(3,3)
    \psset{unit=3}%
    \multido{\nHue=0.01+0.01}{100}{%
        \definecolor[MyColor][H=\nHue,S=1,B=1]
        \pscircle[linewidth=0.01,linecolor=MyColor]{\nHue}}
\endpspicture

\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 TEX grouping mechanism.

```
...
\psset{Param=99}% Global change until the end of the file
...
\bgroup
    \psset{Param=7.5}% Global change inside the defined group
    ...
    \bgroup
        \psset{Param=21}% Global change inside the new group
        ...
        \PstXxxx[Param=-44]% Local change for this macro only
        ...% Param=21
    \egroup
    ...% Param=7.5
\egroup
...% Param=99
```

L.7 Defining parameters using macros

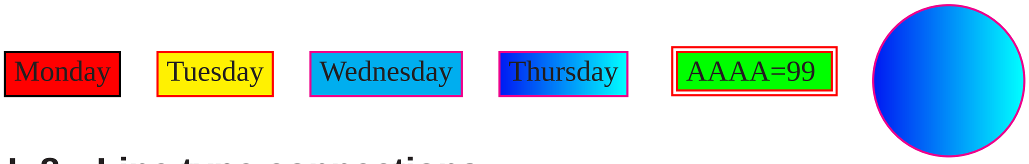
This is not directly possible to define parameters using macros (such behavior is also not supported by the ‘keyval’ package [35]), as in:

```
\def\PstMyParameters{linecolor=red,fillstyle=solid,fillcolor=yellow}
\psset{\PstMyParameters}
\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.

```
% D.G. - 2001 / 2003

% New \PstSet macro
\def\PstSet#1{\PstSet@i#1,\@nil\ignorespaces}
\def\PstSet@i#1,{%
\expandafter\PstSet@ii#1,\@nil
\@ifnextchar\@nil{\@gobble}{\PstSet@i}}
\def\PstSet@ii#1,{%
\PstSet@iii#1=\@nil
\@ifnextchar\@nil{\@gobble}{\PstSet@ii}}
\def\PstSet@iii#1=#2\@nil{\pssetMOD{#1=#2}}

% Redefinition of \psset to accept arguments in macros
\let\@pssetORI\@psset
\def\pssetMOD#1{\@pssetORI#1,\@nil\ignorespaces}
\let\psset\PstSet
\let\@psset\PstSet@i

% For “keyval” and “pst-key” compatibility
\let\KV@setkeysORIG\KV@setkeys

\def\KV@setkeys#1#2{%
\edef\@tempa{#1}%
\edef\@tempb{psset}%
\ifx\@tempa\@tempb
   \PstSet@i#2,\@nil%
\else
   \KV@setkeysORIG{#1}{#2}%
\fi}

% We must restore the original PSTricks definition of \use@par
\def\use@par{%
\ifx\pst@par\@empty\else
\expandafter\@psset\pst@par,\@nil
\def\pst@par{}%
\fi}

% Examples with the “pst-key” interface
```

```
\define@key{psset}{Integer}{\pst@getint{#1}{\PstInteger}}
\define@key{psset}{String}{\edef\PstString{#1}}

\def\ParametersA{fillstyle=solid,fillcolor=red}
\def\ParametersB{linecolor=red,fillstyle=solid,fillcolor=yellow}
\def\ParametersC{linecolor=magenta,fillstyle=solid,fillcolor=cyan}
\def\ParametersD{fillstyle=gradient,gradmidpoint=1,gradangle=90}
\def\ParametersE{Integer=99,String=AAAA}
\def\ParametersF{Integer=1}

\psframebox[\ParametersA]{Monday}
\hfill
\psset{\ParametersB}%
\psframebox{Tuesday}
\hfill
\setkeys{psset}{\ParametersC}%
\psframebox{Wednesday}
\hfill
\psframebox[\ParametersD]{Thursday}
\hfill
\setkeys{psset}{\ParametersE}
\psframebox[\ParametersB,fillcolor=green,doubleline=true]{%
   \PstString=\PstInteger}
\hfill
\pscircle[\ParametersD,\ParametersF]{\PstInteger}
```



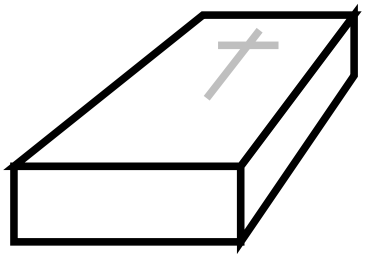
L.8 Line type connections

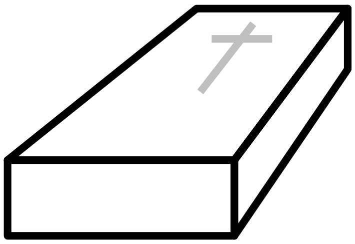
The setlinecap PostScript parameter, which specify how the ends of a line finish, is accessible in PSTricks with the c and C arrows, which put this parameter to 1 and 2, respectively. But other PostScript parameters, and specially the setlinejoin are useful, to change the default behavior for line connections. 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:

```
\psset{linewidth=0.1}
\pspicture(4.5,3)
   \pspolygon(0,0)(0,1)(3,1)(3,0)
   \pspolygon(0,1)(2.5,3)(4.5,3)(3,1)
   \pspolygon(3,0)(3,1)(4.5,3)(4.5,2.2)
   \psline[linecolor=lightgray](2.7,2.6)(3.5,2.6)
   \psline[linecolor=lightgray](2.55,1.9)(3.25,2.8)
\endpspicture
\hfill
\pspicture(4.5,3)
   \pscustom{%
```

```
        \code{1  setlinejoin}
        \pspolygon(0,0)(0,1)(3,1)(3,0)
        \pspolygon(0,1)(2.5,3)(4.5,3)(3,1)
        \pspolygon(3,0)(3,1)(4.5,3)(4.5,2.2)}
    \psline[linecolor=lightgray](2.7,2.6)(3.5,2.6)
    \psline[linecolor=lightgray](2.55,1.9)(3.25,2.8)
\endpspicture
```





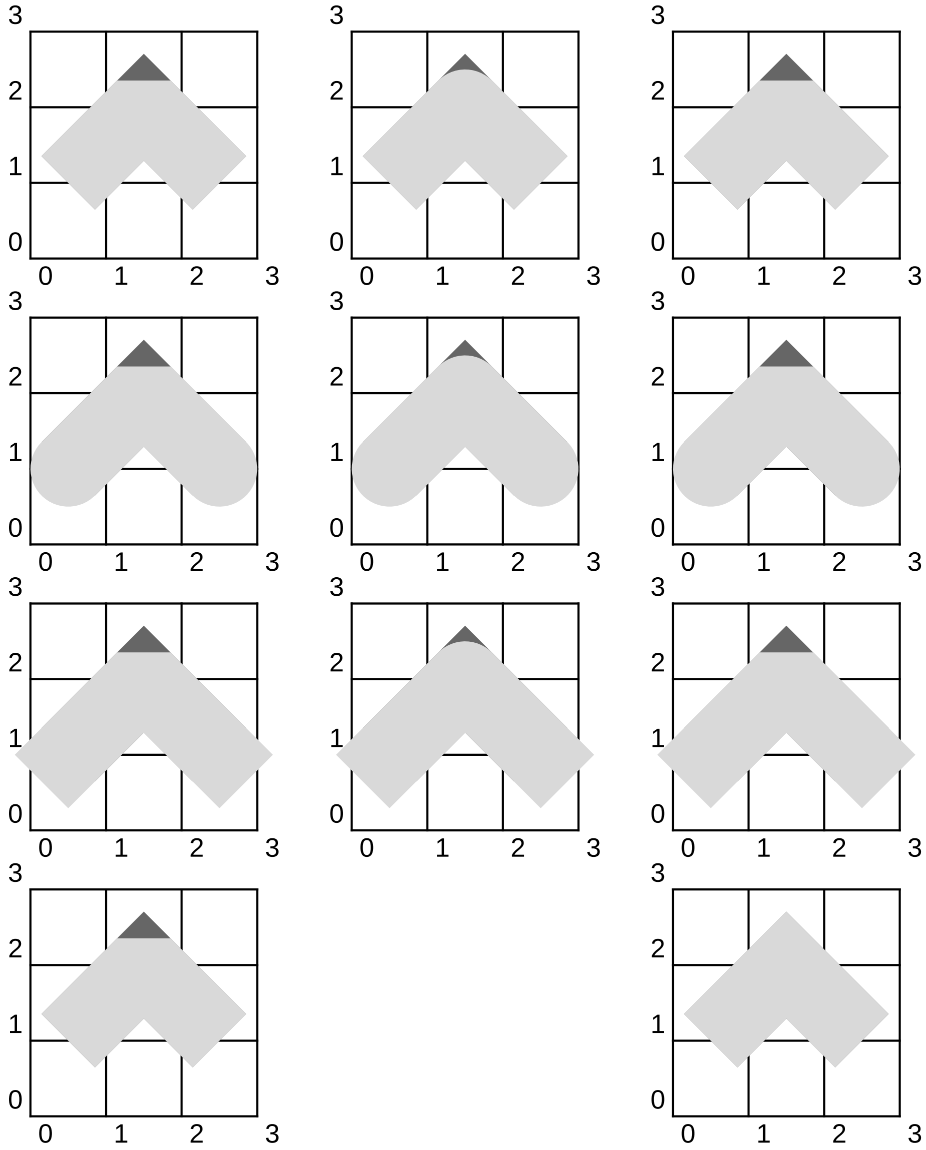
But for intensive usage, it is easier to redefine the PSTricks internal relevant macro (`\psls@solid`, for a line of the `solid` style) and to define the interesting PostScript parameters as PSTricks ones (we add here the `setmiterlimit` one too).[84]

```
% D.G. - May 2003

\def\psset@linecap#1{\pst@getint{#1}{\pslinecap}}
\psset@linecap{0}

\def\psset@linejoin#1{\pst@getint{#1}{\pslinejoin}}
\psset@linejoin{0}

\def\psset@miterlimit#1{\pst@checknum{#1}{\psmiterlimit}}
\psset@miterlimit{10}

\def\psls@solid{%
\pslinejoin\space  setlinejoin
\psmiterlimit\space  setmiterlimit
\pslinecap\space  setlinecap
stroke  }

\def\Test#1#2#3{%
\pspicture(3,3)\psgrid[subgriddiv=0]
    \psset{linewidth=1}%
    \psline[linecolor=DarkGray](0.5,1)(1.5,2)(2.5,1)
    \psline[linecolor=LightGray,linecap=#1,linejoin=#2,miterlimit=#3]
             (0.5,1)(1.5,2)(2.5,1)
\endpspicture}

\Test{0}{0}{1}\hfill\Test{0}{1}{1}\hfill\Test{0}{2}{1}

```

---

[84] To explain the effects of these parameters, we use the same illustration than in the MetaFun manual [40].

```
\Test{1}{0}{1}\hfill\Test{1}{1}{1}\hfill\Test{1}{2}{1}

\Test{2}{0}{1}\hfill\Test{2}{1}{1}\hfill\Test{2}{2}{1}

\Test{0}{0}{1}\hfill\Test{0}{0}{1.5}
```

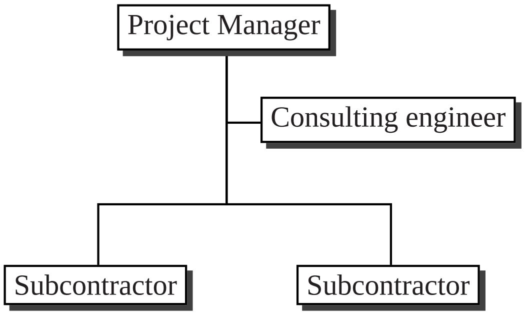


L.9 Boxes of same sizes for trees and diagrams

If we use boxes for entries in a diagram, it could be not very aesthetic to have various different sizes for them.

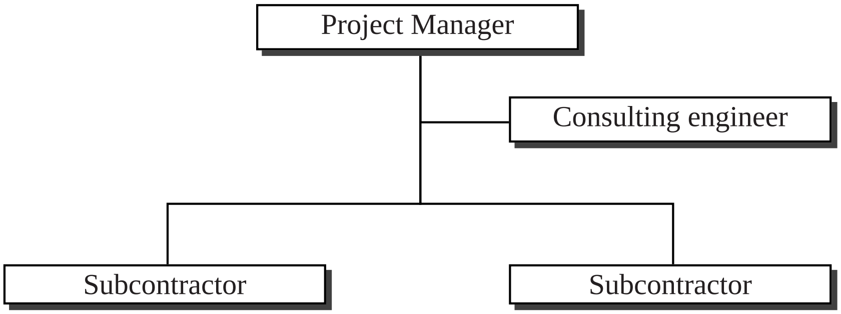
```
\def\MyNode#1{\psshadowbox{#1}}

\psmatrix[mnode=r,colsep=-1]
        &  [name=P]\MyNode{Project  Manager}  \\[-1cm]
        &      &  [name=C]\MyNode{Consulting  engineer}  \\[0pt]
```

```
  [name=S1]\MyNode{Subcontractor} & &
                   [name=S2]\MyNode{Subcontractor}
\endpsmatrix
\ncangle[angleA=-90,angleB=180]{P}{C}
\psset{angleA=-90,angleB=90,armB=0.8}%
\ncangle{P}{S1}
\ncangle{P}{S2}
```



The **\makebox** LaTeX macro (of course, the same result can be obtain otherwise with Plain TeX and ConTeXt) allow to impose that all the boxes will have the same size.

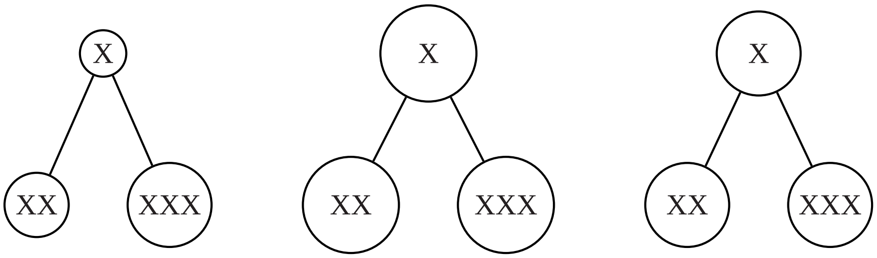
```
% We force the boxes to be of the same size using \makebox
\def\MyNode#1{\psshadowbox{\makebox[4cm]{#1}}}
\psmatrix[mnode=r,colsep=-1]
% ...
```



Obviously, we can do the same thing in other circumstances, as here for trees (we use also a LaTeX syntax to compute the length of the largest label, but this can be done otherwise in Plain TeX and ConTeXt).

```
\def\PstMyNodeA#1{\Tcircle{#1}}
\def\PstMyNodeB#1{\Tcircle{\makebox[1cm]{#1}}}
\newlength{\PstMyLength}
\newcommand{\PstMyNodeC}[1]{\Tcircle{\makebox[\PstMyLength]{#1}}}

\pstree{\PstMyNodeA{X}}{\PstMyNodeA{XX}\PstMyNodeA{XXX}}
\hfill
\pstree{\PstMyNodeB{X}}{\PstMyNodeB{XX}\PstMyNodeB{XXX}}
\hfill
```

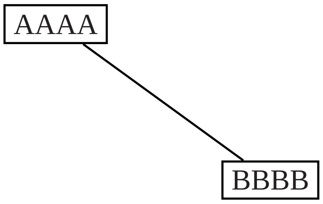
```
% Using LaTeX syntax and the 'calc' package
\settowidth{\PstMyLength}{XXX}%
\pstree{\PstMyNodeC{X}}{\PstMyNodeC{XX}\PstMyNodeC{XXX}}
```

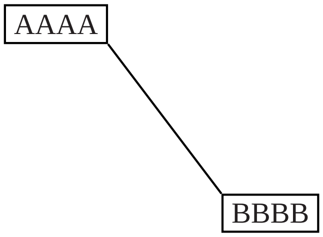


L.10 Reference points

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.

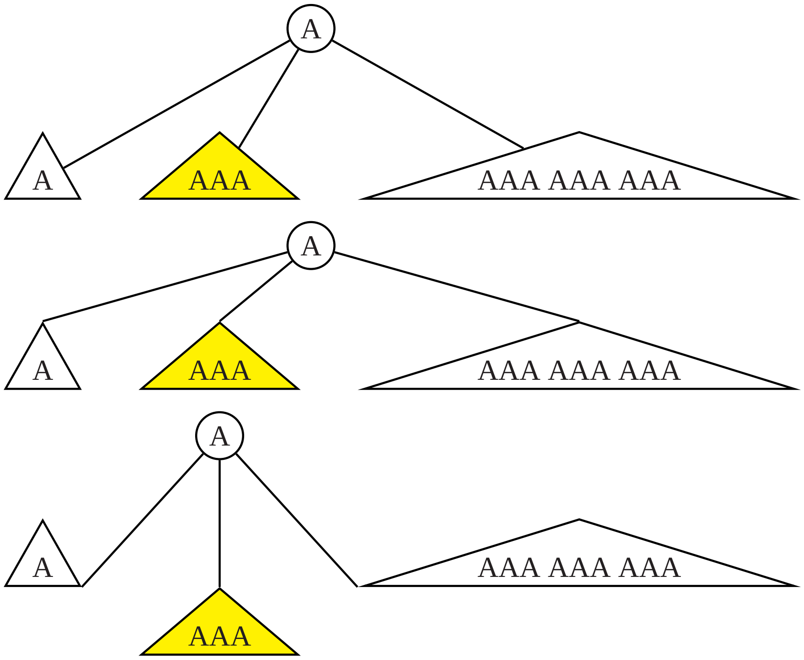
```
\psmatrix[mnode=r]
   [name=A]  \psframebox{AAAA}  &  \\
      &  [name=B]  \psframebox{BBBB}
\endpsmatrix
\ncline{A}{B}
\hfill
\psmatrix[mnode=r]
   [name=A,ref=br]  \psframebox{AAAA}  \\
      &  [name=B,ref=tl]  \psframebox{BBBB}
\endpsmatrix
\ncline{A}{B}
```





```
\pstree{\Tcircle{A}}
         {\Ttri{A}
           \Ttri[fillstyle=solid,fillcolor=yellow]{AAA}
           \Ttri{AAA  AAA  AAA}}

\newcommand{\PstMyTtri}[2][]{\Tr[ref=t]{\pstribox[#1]{#2}}}

\pstree[levelsep=1]{\Tcircle{A}}
         {\PstMyTtri{A}
           \PstMyTtri[fillstyle=solid,fillcolor=yellow]{AAA}
           \PstMyTtri{AAA  AAA  AAA}}
```

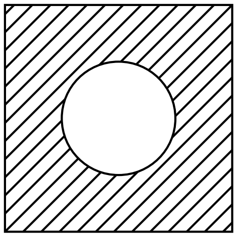
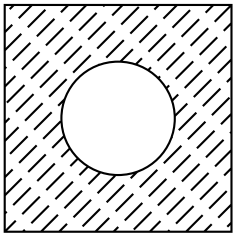
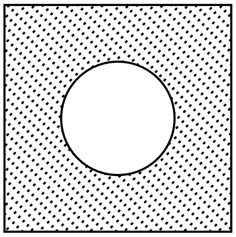
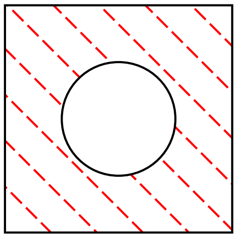
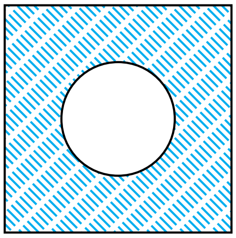
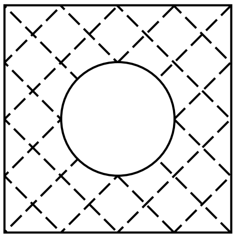
```

\pstree{\Tcircle{A}}
      {\Tr[ref=br]{\pstribox{A}}
       \Tr[ref=t]{\pstribox[fillstyle=solid,fillcolor=yellow]{AAA}}
       \Tr[ref=bl]{\pstribox{AAA  AAA  AAA}}}
```



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).

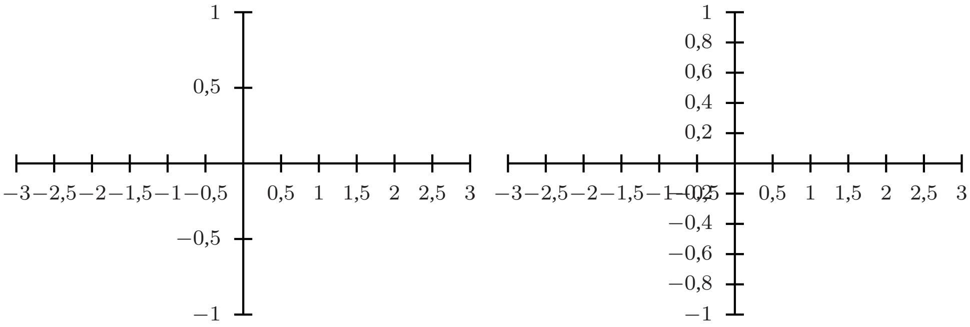
```
% D.G. - September 2002

% To define dashed and dotted hatched styles
\def\pshs@solid{0 setlinecap }
\def\pshs@dashed{[ \psk@dash ] 0 setdash }
\def\psset@hatchstyle#1{%
\@ifundefined{pshs@#1}%
{\@pstrickserr{Hatch style ‘#1’ not defined}\@eha}%
{\edef\pshatchstyle{#1}}}

\psset@hatchstyle{solid}

\def\pst@linefill{%
\@nameuse{pshs@\pshatchstyle}
\psk@hatchangle rotate
\psk@hatchwidth SLW
\pst@usecolor\pshatchcolor
\psk@hatchsep
\tx@LineFill}

\def\TestHatchedStyle#1{%
\pspicture(3,3)
    \psframe[#1](3,3)
    \pscircle[fillstyle=solid,fillcolor=white](1.5,1.5){0.75}
\endpspicture}

\TestHatchedStyle{fillstyle=hlines}
\hfill
\TestHatchedStyle{fillstyle=hlines,hatchstyle=dashed,dash=0.3  0.15}
\hfill
\TestHatchedStyle{fillstyle=hlines,hatchstyle=dashed,dash=0.01  0.10,
                              hatchsep=0.05}

\TestHatchedStyle{fillstyle=vlines,hatchstyle=dashed,
            hatchsep=0.4,hatchcolor=red}
\hfill
\TestHatchedStyle{fillstyle=vlines,hatchstyle=dashed,
            hatchsep=0.05,dash=0.2  0.1,hatchcolor=cyan}
\hfill
\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 separated 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’).

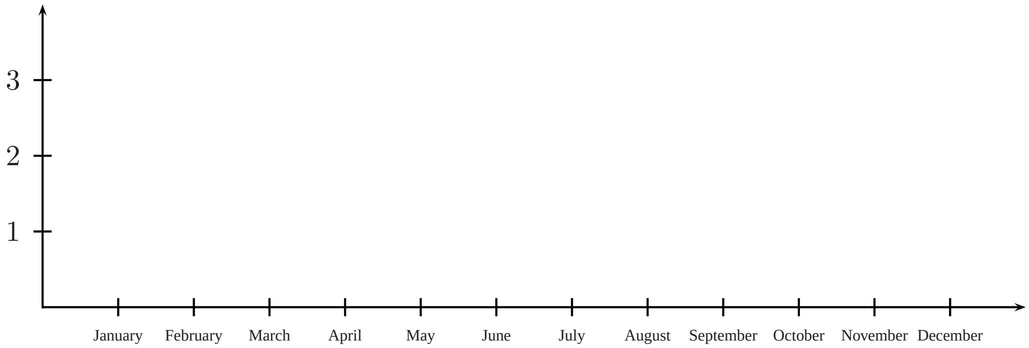
```
% D.G. - June 1997

\def\PstLabelComma#1.#2.#3\@nil{%
\ifx#1\@empty$#1$\else$#1$\fi
\ifx#2\@empty\else\ifnum#2=0\else,$#2$\fi\fi}

% To change decimal points by commas in axe labels of \psaxes
\def\pshlabel#1{\expandafter\PstLabelComma#1..\@nil}
\def\psvlabel#1{\expandafter\PstLabelComma#1..\@nil}

\scriptsize

\pspicture(-3,-2.2)(3,2.2)
  \psaxes[Dx=0.5,Dy=0.5,yunit=2](0,0)(-3,-1)(3,1)
\endpspicture
\hfill
\pspicture(-3,-2.2)(3,2.2)
  \psaxes[Dx=0.5,Dy=0.2,yunit=2](0,0)(-3,-1)(3,1)
\endpspicture
```

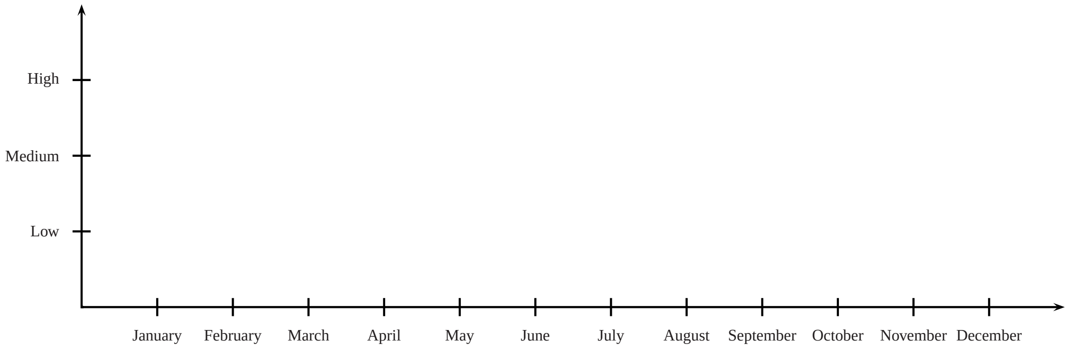


L.13 Alphabetic labels on axes

This is easy to use alphabetic labels rather that numeric ones on axes, redefining the **\pshlabel** and **\psvlabel** macros and using the ‘arrayjob’ package [4].

```
\newarray{\Months}
\readarray{Months}{January&February&March&April&May&June&%
July&August&September&October&November&December}

\newarray{\Levels}
\readarray{Levels}{Low&Medium&High}

\def\pshlabel#1{\tiny\Months(#1)}
\psaxes[showorigin=false]{->}(13,4)

\def\psvlabel#1{\tiny\Levels(#1)}
\psaxes[showorigin=false]{->}(13,4)
```

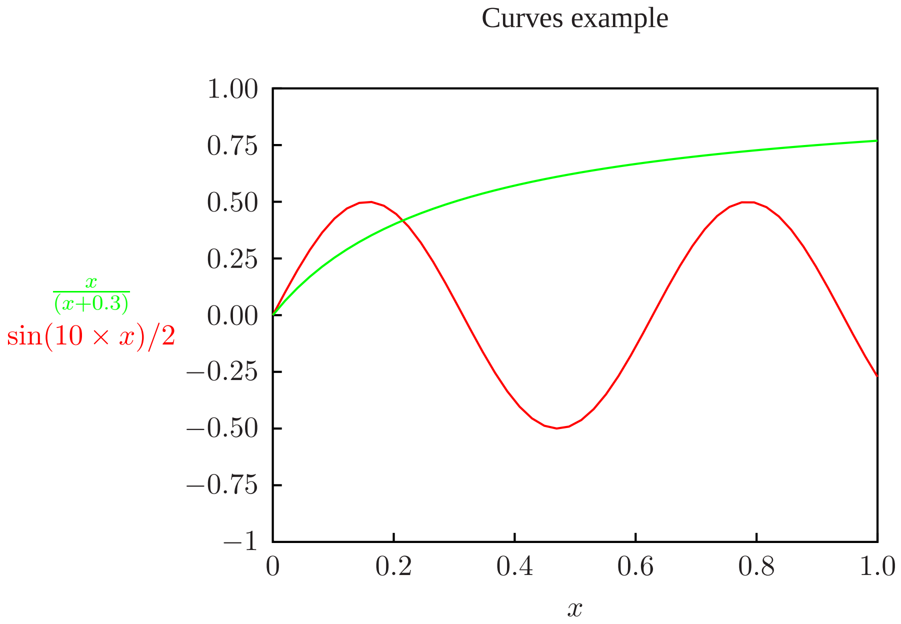


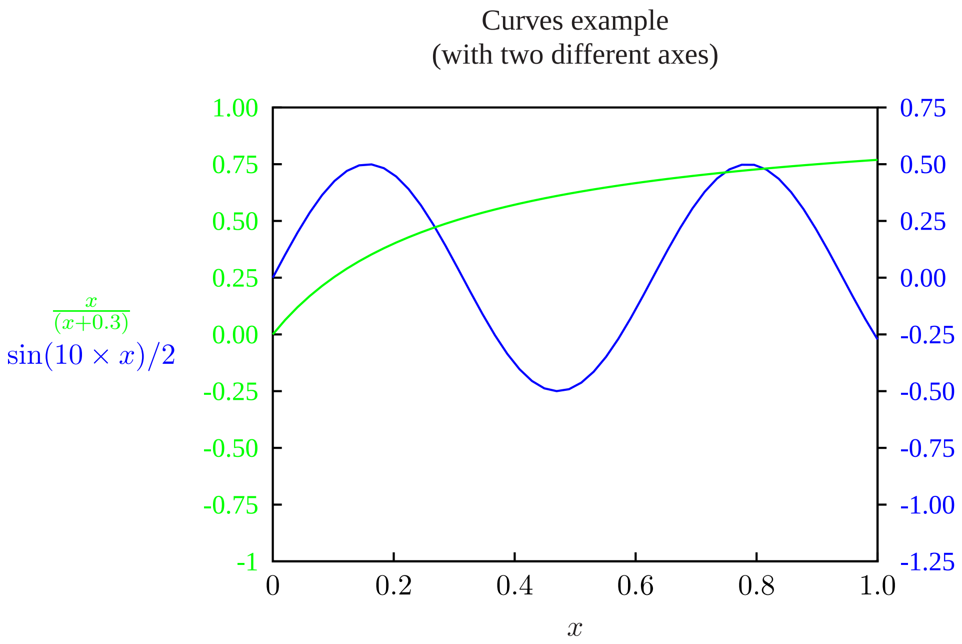


L.14 Two different axes

This is also not difficult to put two different axes on the same graph.

```
\psset{xunit=8,yunit=3}%

\pspicture(-0.5,-1.5)(1,1.5)
  \psaxes[Dx=0.2,Oy=-1,Dy=0.25,tickstyle=top,
          axesstyle=frame](0,-1)(1,1)
  \rput(0.5,1.3){Curves example}
  \rput(-0.3,0){%
    \shortstack{\textcolor{green}{$\frac{x}{(x + 0.3)}$}\\
                \textcolor{red}{$\sin (10 \times x) / 2$}}}
  \rput(0.5,-1.3){$x$}
  \psplot[linecolor=red]% 1 radian = 57.296 degrees
      {0}{1}{x 10 mul 57.296 mul sin 0.5 mul}% sin(10 x) / 2
  \psplot[linecolor=green]{0}{1}{x 0.3 x add div}% x / (x + 0.3)
\endpspicture

% Same curves, but with different origin and axe for the first plot
\pspicture(-0.5,-1.5)(1,1.5)
  \def\psvlabel#1{\small\textcolor{green}{#1}}%
  \psaxes[Dx=0.2,Oy=-1,Dy=0.25,tickstyle=top,
          axesstyle=frame](0,-1)(1,1)
  \def\psvlabel#1{\small\textcolor{blue}{#1}}%
  \psaxes[Oy=-1.25,Dy=0.25,tickstyle=bottom,
          ticks=y,labels=y](1,-1)(0,1)
  \rput(0.5,1.3){%
    \shortstack{Curves example\\(with two different axes)}}
  \rput(-0.3,0){%
    \shortstack{\textcolor{green}{$\frac{x}{(x + 0.3)}$}\\
             \textcolor{blue}{$\sin (10 \times x) / 2$}}}
  \rput(0.5,-1.3){$x$}
  \rput(0,0.25){%
    \psplot[linecolor=blue]% 1 radian = 57.296 degrees
        {0}{1}{x 10 mul 57.296 mul sin 0.5 mul}}% sin(10 x) / 2
  \psplot[linecolor=green]{0}{1}{x 0.3 x add div}% x / (x + 0.3)
\endpspicture
```





DG: Add examples of logarithmic axes and different tick styles, without the too long code.

L.15 Polar Plots

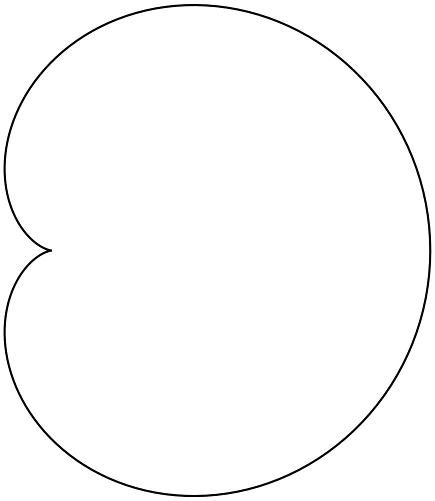
Polar plots can be done with a small modification of the **\psplot** macro.[85]

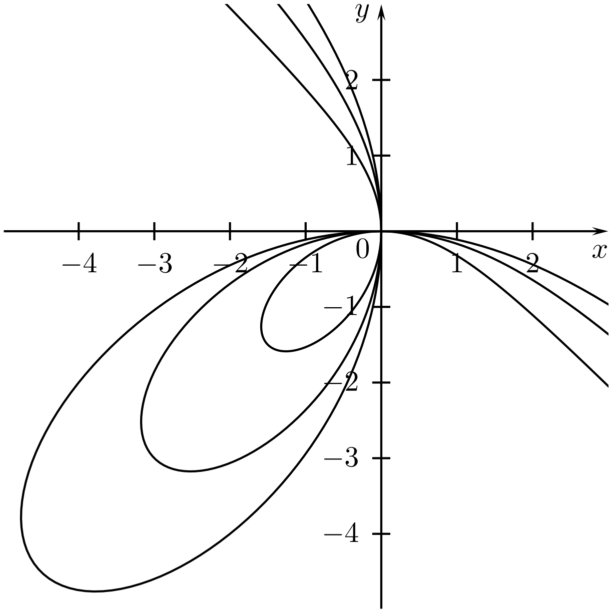
```
% D.G. - June 1998 (according the way suggested by Ulrich Dirr)

% For polar plots
\newif\ifpolarplot
\def\psset@polarplot#1{\@nameuse{polarplot#1}}
\psset@polarplot{false}

\def\psplot@i#1#2#3{%
\pst@killglue
\begingroup
  \use@par
  \@nameuse{beginplot@\psplotstyle}%
  \ifpolarplot
    \addto@pscode{%
      \psplot@init
      /x #1 def
      /x1 #2 def
      /dx x1 x sub \psk@plotpoints div def
      /xy {% Adapted from \parametricplot@i
        #3 dup x cos mul exch x sin mul
        \pst@number\psxunit mul exch
```

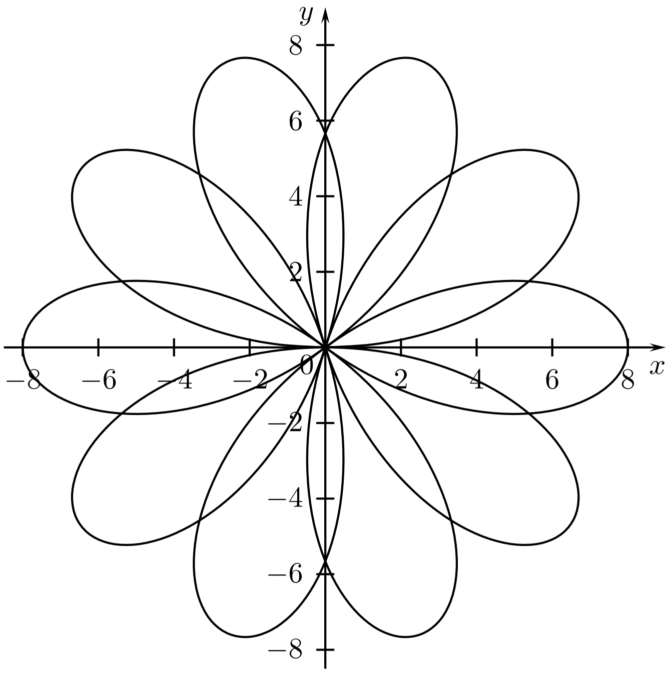
---

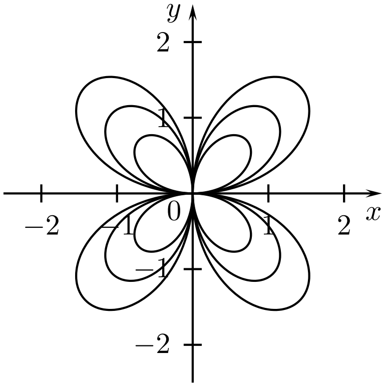
[85] This code was implemented in 1998 according the way suggested by Ulrich Dirr. And the examples 2, 3 and 4 come also from him.

```
        \pst@number\psyunit mul exch
      } def}%
  \else
    \addto@pscode{%
      \psplot@init
      /x #1 def
      /x1 #2 def
      /dx x1 x sub \psk@plotpoints div def
      /xy {
        x \pst@number\psxunit mul
        #3 \pst@number\psyunit mul
      } def}%
  \fi
  \gdef\psplot@init{}%
  \@pstfalse
  \@nameuse{testqp@\psplotstyle}%
  \if@pst
    \psplot@ii
  \else
    \psplot@iii
 \fi
\endgroup
\ignorespaces}

\psset{plotpoints=200}%

\pspicture(-0.6,-3.3)(5,3.3)
  % Cardioide
  \psplot[polarplot=true,unit=0.5]{0}{360}{x cos 1 add 5 mul}
\endpspicture
\hfill
\pspicture*(-5,-5)(3,3)
  \psaxes[arrowlength=2]{->}(0,0)(-4.99,-4.99)(3,3)
  \rput[Br](3,-0.35){$x$}
  \rput[tr](-0.15,3){$y$}
  \rput[Br](-0.15,-0.35){$0$}
  \psset{polarplot=true}%
  \psplot{140}{310}{3 neg x sin mul x cos mul
                    x sin 3 exp x cos 3 exp add div}
  \psplot{140}{310}{6 neg x sin mul x cos mul
                    x sin 3 exp x cos 3 exp add div}
  \psplot{140}{310}{9 neg x sin mul x cos mul
                    x sin 3 exp x cos 3 exp add div}
\endpspicture

\pspicture(-2.5,-2.5)(2.5,2.5)
  \psaxes[arrowlength=2]{->}(0,0)(-2.5,-2.5)(2.5,2.5)
  \rput[Br](2.5,-0.35){$x$}
  \rput[tr](-0.15,2.5){$y$}
  \rput[Br](-0.15,-0.35){$0$}
  \psset{plotstyle=curve,polarplot=true}%
  \psplot{0}{360}{x cos 2 mul x sin mul}
  \psplot{0}{360}{x cos 3 mul x sin mul}
  \psplot{0}{360}{x cos 4 mul x sin mul}
\endpspicture
\hfill
\psset{unit=0.5}%
\pspicture(-8.5,-8.5)(9,9)
  \psaxes[Dx=2,dx=2,Dy=2,dy=2,arrowlength=2]{->}(0,0)(-8.5,-8.5)(9,9)
  \rput[Br](9,-0.7){$x$}
  \rput[tr](-0.3,9){$y$}
  \rput[Br](-0.3,-0.7){$0$}
  \psplot[plotstyle=curve,polarplot=true]{0}{720}{8 2.5 x mul sin mul}
\endpspicture
```



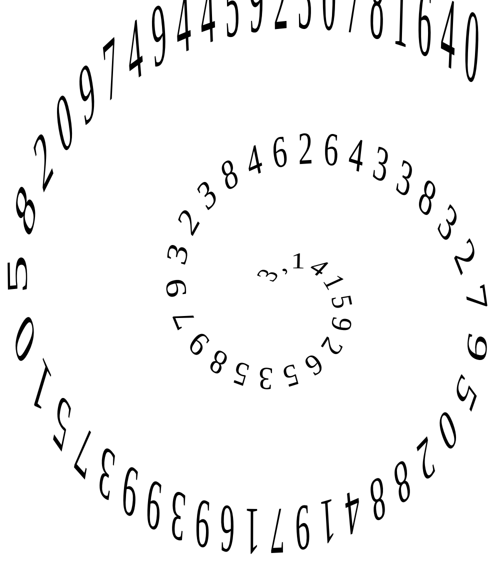






Another example, using both a polar plot and putting a text around it, with the **\pstextpath** macro. As we also want to slightly increase the size of the characters along the ellipse, we use a generic macro from Juergen Schlegelmilch to get one character at a time and apply an action on it).

```
% A \DoPerCharacter macro adapted from Juergen Schlegelmilch
% (posted on comp.text.tex - January 1998)
\def\DoPerCharacter#1#2#3\@nil{%
#1#2%
\edef\@tempa{#3}%
\ifx\@tempa\empty
\else
  \DoPerCharacter#1#3\@nil
\fi}

\def\PerCharacter#1#2{\DoPerCharacter#1#2\@nil}

\pst@dimh=0.7pt

\def\CharacterAction#1{%
\scalebox{1 \pst@number{\pst@dimh}}{#1}
\advance\pst@dimh by 0.04pt}

\pstextpath{\psplot[linestyle=none,polarplot=true,plotpoints=300,unit=5]
    {500}{-500}{1 2.7182818 x 200 div exp 1 add div}}% Spiral
  {\PerCharacter{\CharacterAction}{%
      3,14159265358979323846264338327950288419716939937510%
         582097494459230781640}}
```

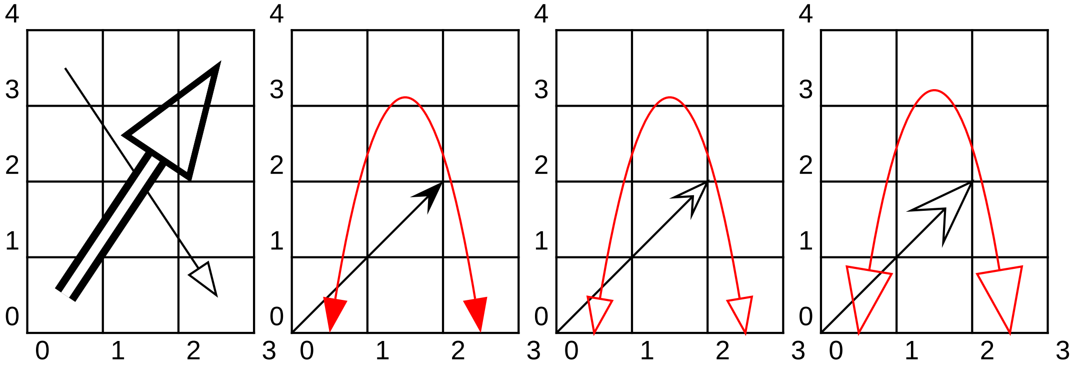


L.16 Non filled arrows

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.)

```
% D.G. - July 1999

% Redefinition of \psset@arrowscale to store the value
% of the X scale factor
\def\psset@arrowscale#1{%
\psset@arrowscale@i#1 \@nil
\pst@getscale{#1}\psk@arrowscale}

\def\psset@arrowscale@i#1 #2\@nil{\edef\pst@arrowscale{#1}}

\def\pst@arrowscale{1}% Default value

% New parameter \newif\ifpsarrowfill
\def\psset@arrowfill#1{\@nameuse{psarrowfill#1}}
\psset@arrowfill{true}

% Modification of the PostScript macro “Arrow” to choose to fill
% or not the arrow (it require to restore the current linewidth,
% despite of the scaling)
\pst@def{Arrow}<{%
CLW mul add dup 2 div
/w ED mul dup
/h ED mul
/a ED
{ 0 h T 1 -1 scale } if
gsave
\ifpsarrowfill
\else
   \pst@number{\pslinewidth} \pst@arrowscale\space div
   \ifpsdoubleline\space 4 div\fi\space
   SLW
\fi
w neg h moveto
0 0 L w h L w neg a neg rlineto
\ifpsarrowfill
   fill
\else
   closepath stroke
\fi
grestore
0 h a sub moveto}>

\psset{subgriddiv=0}%

\pspicture(3,4)\psgrid
   \psset{arrowfill=false,arrows=->,arrowinset=0}%
   \psline[arrowsize=0.3](0.5,3.5)(2.5,0.5)
```

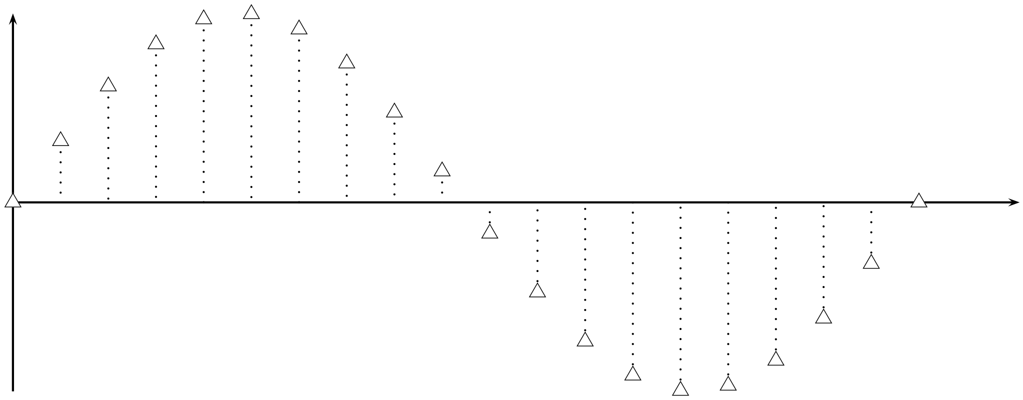
```
    \pnode(0.5,0.5){A}
    \pnode(2.5,3.5){B}
    \ncline[linewidth=0.1,arrowsize=1,doubleline=true]{A}{B}
\endpspicture
\hfill
\psset{subgriddiv=0,arrowscale=3}%
\pspicture(3,4)\psgrid
    \psline{->}(2,2)
      \parabola[linecolor=red,arrowinset=0]{<->}(0.5,0)(1.5,3)
\endpspicture
\hfill
\pspicture(3,4)\psgrid
    \psset{arrowfill=false}%
    \psline{->}(2,2)
    \parabola[linecolor=red,arrowinset=0]{<->}(0.5,0)(1.5,3)
\endpspicture
\hfill
\pspicture(3,4)\psgrid
    \psset{arrowfill=false,arrowsize=0.2}%
    \psline{->}(2,2)
    \parabola[linecolor=red,arrowinset=0]{<->}(0.5,0)(1.5,3)
\endpspicture
```

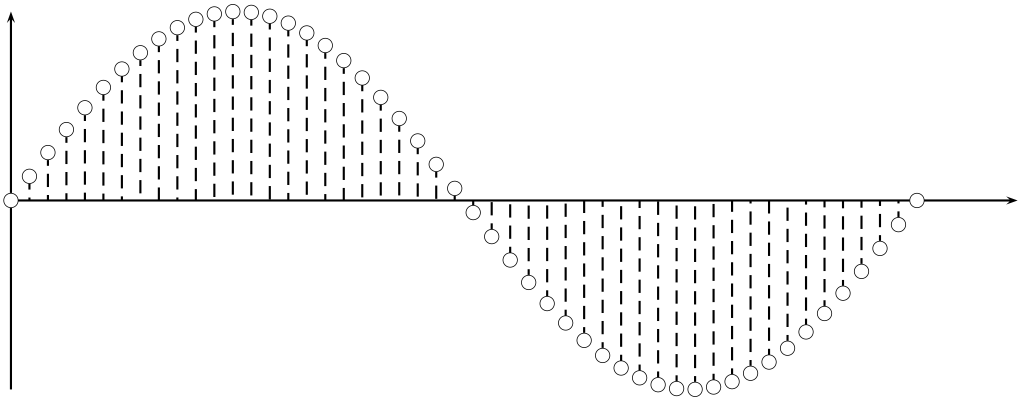


L.17 Lines under a curve

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.

```
% D.G. - January 2000

\let\beginplot@LineToXAxis\beginplot@line
\def\endplot@LineToXAxis{\psLineToXAxis@ii}
\let\beginqp@LineToXAxis\beginqp@line
\let\doqp@LineToXAxis\doqp@line
\let\endqp@LineToXAxis\endqp@line
\let\testqp@LineToXAxis\testqp@line

\def\psLineToXAxis@ii{%
\addto@pscode{\pst@cp  \psline@iii  \tx@LineToXAxis}%
\end@OpenObj}

\def\tx@LineToXAxis{LineToXAxis }

% Adapted from the Line PostScript macro
\pst@def{LineToXAxis}<{%
NArray
n 0 eq not
  { n 1 eq { 0 0 /n 2 def } if
    ArrowA
    /n n 2 sub def
    CP 2 copy moveto pop 0 Lineto
    n { 2 copy moveto pop 0 Lineto } repeat
    CP
    4 2 roll
    ArrowB
    2 copy moveto pop 0
    L
    pop pop } if}>

\psset{xunit=0.0333,yunit=2.5}

\pspicture(0,-1)(400,1)
  \psline{->}(0,0)(400,0)
  \psline{->}(0,-1)(0,1)
  \psplot[plotstyle=LineToXAxis,plotpoints=20,linestyle=dotted,
        showpoints=true,dotstyle=triangle,dotsize=0.2]{0}{360}{x sin}
\endpspicture

\pspicture(0,-1)(400,1)
  \psline{->}(0,0)(400,0)
  \psline{->}(0,-1)(0,1)
  \psplot[plotstyle=LineToXAxis,plotpoints=50,linestyle=dashed,
        showpoints=true,dotstyle=o,dotsize=0.2]{0}{360}{x sin}
\endpspicture
```



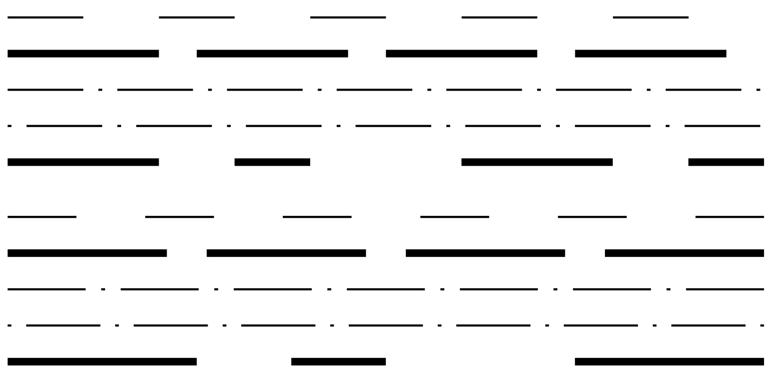


L.18 Dashed lines

We can extend the dashed **linestyle** to accept dashes defined by two different segments.

```
% D.G. - November 1997

\def\psset@dash#1{%
\pst@expandafter\psset@@dash{#1} {\z@} {\z@}
   {\pst@missing} {\pst@missing} {}\@nil

\edef\psk@dash{%
\pst@number\pst@dimg \pst@number\pst@dimh
\pst@number\pst@dimc \pst@number\pst@dimd}}

\def\psset@@dash#1 #2 #3 #4 #5\@nil{%
\pssetlength{\pst@dimg}{#1}%
\pssetlength{\pst@dimh}{#2}%
\pssetlength{\pst@dimc}{#3}%
\pssetlength{\pst@dimd}{#4}}

\psset@dash{5pt 3pt}% Default value

\pst@def{DashLine}<%
dup 0 gt
   { /a .5 def PathLength exch div }
   { pop /a 1 def PathLength }
   ifelse
/b ED
/x1 ED
/y1 ED
/x ED
/y ED
/z y x add y1 add x1 add def
/Coef b a .5 sub 2 mul y mul sub z Div round
       z mul a .5 sub 2 mul y mul add b exch Div def
/y y Coef mul def
/x x Coef mul def
/y1 y1 Coef mul def
```

```
/x1 x1 Coef mul def
x1 0 gt y1 0 gt x 0 gt y 0 gt and
    { [ y x y1 x1 ] 1 a sub y mul}
    { [ 1 0] 0 }
    ifelse
setdash
stroke>

\def\TestDashedLine[#1]{%
\pspicture(0,-0.08)(10,0.08)
  \psline[#1](10,0)
\endpspicture}

\psset{dashadjust=false}%
\TestDashedLine[linestyle=dashed,dash=1 1]
\TestDashedLine[linestyle=dashed,linewidth=1mm,dash=2 0.5]
\TestDashedLine[linestyle=dashed,dash=1 0.2 0.05 0.2]
\TestDashedLine[linestyle=dashed,dash=0.05 0.2 1 0.2]
\TestDashedLine[linestyle=dashed,linewidth=1mm,dash=2 1 1 2]

\psset{dashadjust=true}%
\TestDashedLine[linestyle=dashed,dash=1 1]
\TestDashedLine[linestyle=dashed,linewidth=1mm,dash=2 0.5]
\TestDashedLine[linestyle=dashed,dash=1 0.2 0.05 0.2]
\TestDashedLine[linestyle=dashed,dash=0.05 0.2 1 0.2]
\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' LaTeX packages

The 'color' [7] LaTeX package (also usable in PLAIN TeX) is the way to use to define and manage colors in LaTeX and PLAIN TeXsince the arrival of LaTeX2e in 1993. Nevertheless, it is not stricly 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] LaTeX 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 LaTeX always use the ‘pstcol’ package.

M.2 Compatibility problem between PSTricks and the ‘graphicx’ LaTeX package for the \scalebox macro

Both PSTricks and the ‘graphicx’ [27] LaTeX 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.

```
% D.G. - March 2002 and July 2003

\usepackage{multido}

% Compatibility clash between ‘fp’ and ‘multido’
% when using \n.. multido indexes
% (to add after ‘multido’ and before ‘fp’)
\let\FPaddMultiDo\FPadd
%
\def\FPsub#1#2{%
\edef\multido@temp{\noexpand\FPsub@#2\noexpand\@empty}%
\FPaddMultiDo{#1}{\multido@temp}}
%
\let\FPsubMultiDo\FPsub
%
\def\multido@init@n#1#2#3{%
\edef#3{#1}%
\ifnum\multido@count<\z@
   \expandafter\FPsubMultiDo
\else
   \expandafter\FPaddMultiDo
\fi
{0}{#2}\multido@temp
\multido@addtostep{\do\FPaddMultiDo{\do#3}{\multido@temp}{\do#3}}}

\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 Zhuhan Jiang. CTAN:macros/generic/arrrayjob

[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 LaTeX, by Timothy van Zandt. CTAN:macros/latex/contrib/fancybox

[16] ‘fancyvrb’. Fancy Verbatims in LaTeX, by Timothy van Zandt. CTAN:macros/latex/contrib/fancyvrb

[17] Gabriel Valiente Feruglio, *Typesetting commutative diagrams*, TUGBoat, Volume 15, Number 4, pages 466–484, 1994.

[18] ‘fltpoint’. Floating point arithmetic with TeX, by Eckhart Guthölrhein. CTAN:macros/latex/contrib/fltpoint

[19] ‘fp’. Fixed point arithmetic for TeX, 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 LaTeX 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 LaTeX Graphics, by David Carlisle and Sebastian Rahtz. CTAN:macros/latex/graphics/graphicx.dtx

[28] Branko Grünbaum and Geoffrey Shephard. *Tilings and Patterns*, Freeman and Company, 1987.

[29] Eitan M. Gurari, *TeX and LaTeX: Drawing and Literate Programming*, McGraw-Hill, 1994.

[30] Alan Hoenig, *TeX Unbound: LaTeX & TeX 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://www.imagemagick.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 TeXbook*, 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 TeX and LaTeX, 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 TeX, 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] pdfTeX. Variant of the TeX program, by Hàn Thế Thành. See http://tug.org/applications/pdftex

[48] pdfTricks. PSTricks support in PDF with pdfTEX, 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 LATEX documents, by David Kastrup. CTAN:macros/latex/contrib/preview See http://preview-latex.sourceforge.net

[53] ‘ps4pdf’. Usage of PostScript commands inside a pdfLATEX 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 TEX Users Group) See http://sarovar.org/projects/pstricks and http://www.tug.org.in/tutorial/pstricks

[68] PSTricks Web pages, maintened by Denis Girou. See http://www.tug.org/applications/PSTricks

[69] PSTricks Web pages, maintened by Manuel Luque (in French). See http://members.aol.com/Mluque5130

[70] PSTricks Web pages on the Syracuse site, maintened by Jean-Michel Sarlat (in French). See http://melusine.eu.org/syracuse/pstricks

[71] PSTricks Web pages, maintened by Herbert Voß. See http://www.pstricks.de

[72] ‘random’. Generating “random” numbers in TEX, 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 LATEX2e*, by Keith Reickdahl, December 1997, available on CTAN:info/epslatex.pdf

[76] Denis Roegel, *MetaObj: Very High-Level Objects in MetaPost*, TUGBoat, Volume 23, Number 1, pages 93–100, 2002.

[77] David Salomon, *The Advanced TEXbook*, Springer-Verlag, 1995.

[78] ‘Seminar’. A LATEX 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 LATEX), by Sylvain Lombardy and Sylvain Lombardy. See http://www.liafa.jussieu.fr/~lombardy/Vaucanson-G

[81] Herbert Voß and Laura E. Jackson, *Die mathemtischen Funktionen von PostScript*, Die TeXnische Komödie, Volume 1/02, pages 40–47, March 2002 (in german).

[82] VTEX. TEX 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 TEX, by Kristoffer H. Rose and Ross Moore. CTAN:macros/latex/contrib/xypic See also http://www.tug.org/applications/Xy-pic

[84] Hao Wang, *Games, Logic and Computers*, Scientific American, pages 98–106, 1965.