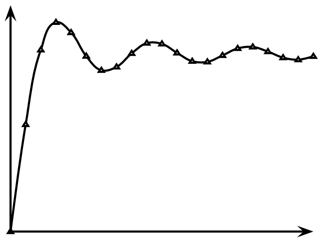
Here is a plot of Integral(sin(x)). The data was generated by Mathematica, with

```
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:

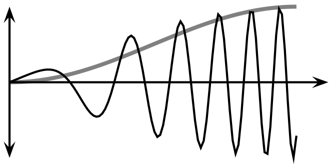
```
...
\listplot[plotstyle=curve,showpoints=true,
  dotstyle=triangle]{\mydata}
...
```

\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 $\sin^2(x)$:

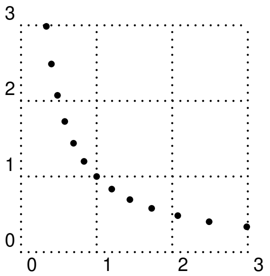


```
\psset{xunit=1.2pt}
\psplot[linecolor=gray,linewidth=1.5pt,plotstyle=curve]%
   {0}{90}{x sin dup mul}
\psplot[plotpoints=100]{0}{90}{x sin x 2 div 2 exp cos mul}
\psline{<->}(0,-1)(0,1)
\psline{->}(100,0)
```

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

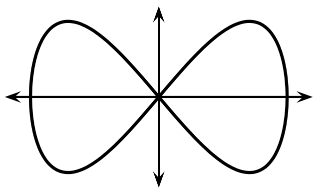
For example,




```
\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

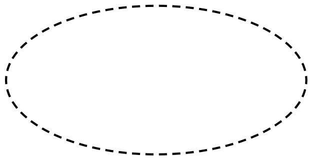
The following graphics parameters (in addition to **linewidth** and **linecolor**) determine how the lines are drawn, whether they be open or closed curves.

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

Valid styles are none, solid, dashed and dotted.

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

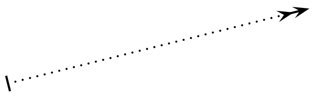
The *black-white* dash pattern for the dashed line style. For example:



```
\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)
```

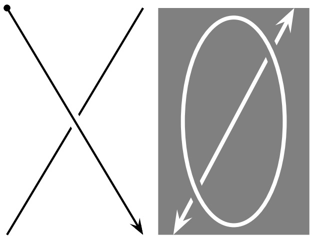
border=*dim* **Default: 0pt**

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.

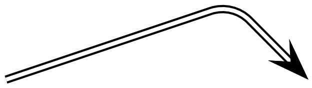
doublesep=*dim* **Default: 1.25\pslinewidth**

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.

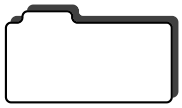
shadowangle=*angle* **Default: -45**

See **shadow**, above.

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

See **shadow**, above.

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

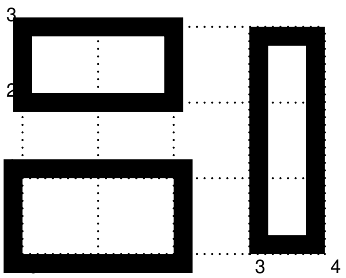


```
\pspolygon[linearc=2pt,shadow=true,shadowangle=45,
   xunit=1.1](-1,-.55)(-1,.5)(-.8,.5)(-.8,.65)
   (-.2,.65)(-.2,.5)(1,.5)(1,-.55)
```

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 and crosshatch*.

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.

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.

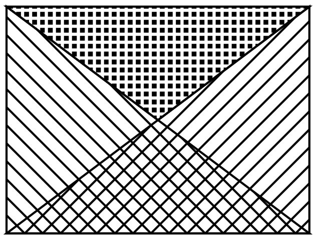
hatchcolor=*color* **Default: black**

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:



```
\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)
```

Don't be surprised if the checkered part of this example (the last **\pspolygon**) looks funny on low-resolution devices. PSTricks adjusts the lines so that they all have the same width, but the space between them, which in this case is black, can have varying width.

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: -**

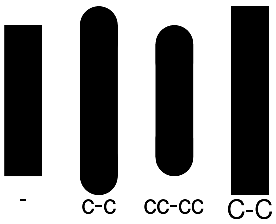
parameter determines what you get. It can have the following values, which are pretty intuitive:[5]

[5]This is TEX's version of WYSIWYG.

Value	*Example*	*Name*
-		None
<->		Arrowheads.
>-<		Reverse arrowheads.
<<->>		Double arrowheads.
>>-<<		Double reverse arrowheads.
|-|		T-bars, flush to endpoints.
|*-|*		T-bars, centered on endpoints.
[-]		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 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: 2pt 3**

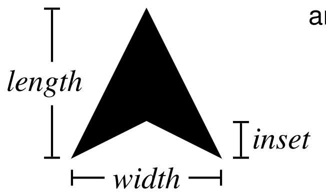
Width of arrowheads, as shown below.

arrowlength=*num* **Default: 1.4**

Length of arrowheads, as shown below.

arrowinset=*num* **Default: .4**

Size of inset for arrowheads, as shown below.



arrowsize	=	*dim num*
width	=	*num* x **linewidth** + *dim1*
length	=	**arrowlength** x *width*
inset	=	**arrowinset** x *height*

tbarsize=*dim num* **Default: 2pt 5**

The width of a t-bar, square bracket or rounded bracket is *num* times **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.

dotsize=*dim num* **Default: .5pt 2.5**

The diameter of a circle or disc is *num* times **linewidth**, plus *dim*.

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.

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 **\psfill** and **\pstroke** (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.

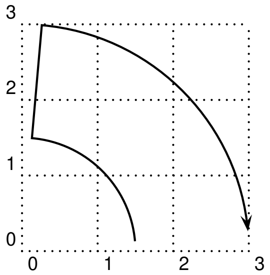
There are three types of graphics objects:

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.[6] 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

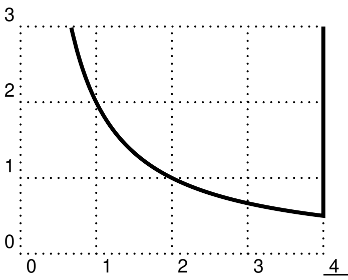



```
\pscustom{
  \psarc(0,0){1.5}{5}{85}
  \psarcn{->}(0,0){3}{85}{5}}
```

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.

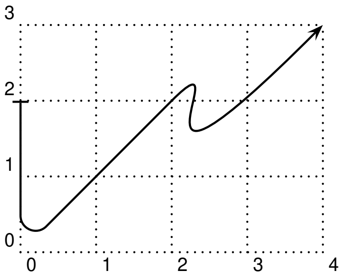
For example:




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

[6]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.

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



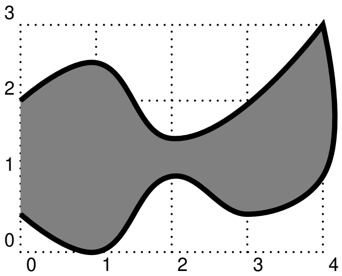

```
\pscustom{
  \psline[linearc=.2]{|-}(0,2)(0,0)(2,2)
  \psbezier{->}(3,3)(1,0)(4,3)}
```

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

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

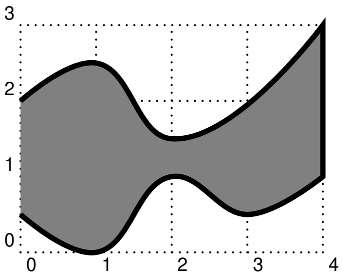
parameter.

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



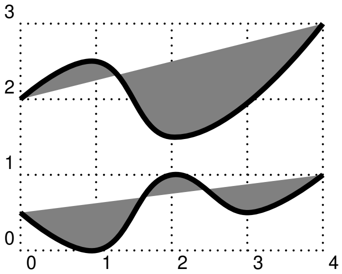

```
\pscustom[linewidth=2pt,fillstyle=solid,fillcolor=gray]{
  \pscurve(0,2)(1,2.5)(2,1.5)(4,3)
  \pscurve(4,1)(3,0.5)(2,1)(1,0)(0,.5)}
```

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*)

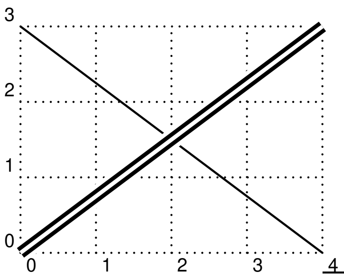
This moves the current point to (x, y).

\closepath

This closes the path, joining the beginning and end of each piece (there may be more than one piece if you use **\moveto**).[7]

\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]}
```

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

\fill[*par*]

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

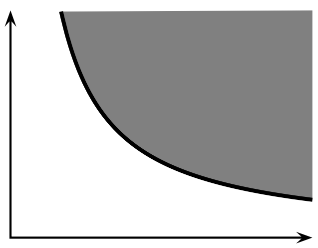
\gsave

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.

\grestore

See above.

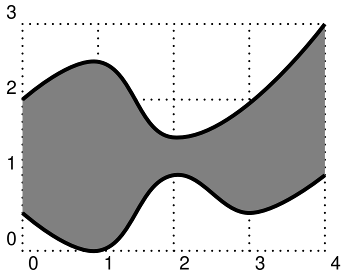
Here is an example that fixes an earlier example, using **\gsave** and **\grestore**:



```
\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 C 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*}

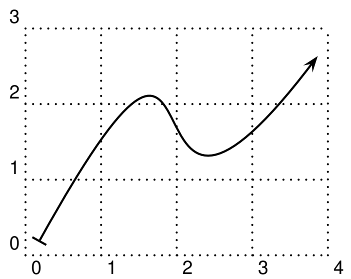
This defines the PostScript operators ArrowA and ArrowB so that

```
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, 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.[8]

LaTeX users can type

[8] 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

\begin{pspicture} ... \end{pspicture}

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>}

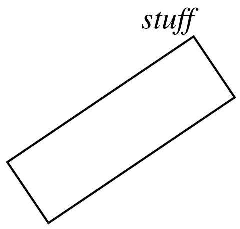
thereby blotting out whatever is behind *stuff*. This is useful for positioning text on top of something else.

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.

---

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.)

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){\em 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

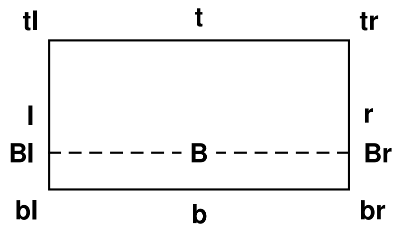
\rput*[*refpoint*]{*rotation*}**(*x,y*){*stuff*}**

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:

Here is a marginal note.

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

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

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

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

The default value of *labelsep* is the dimension register

\pslabelsep

You can also change this be setting the

labelsep=*dim* **Default: 5pt**

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

Here is a simple example:

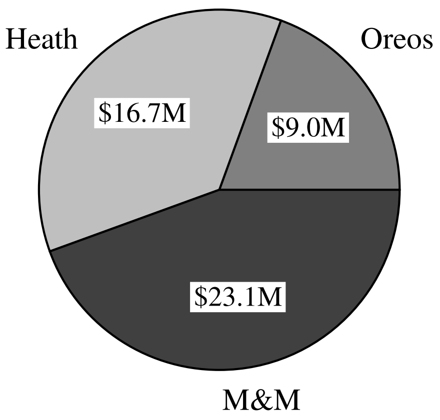
(1,1)

```
\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:[9]

[9] PSTricks is distributed with a useful tool for converting data to piecharts: piechart.sh. This is a UNIX sh script written by Denis Girou.

```
\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
```



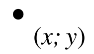
You can use the following abbreviations for *refangle*, which indicate the direction the angle points:[10] [11]

---

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

[11] 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){$(x,y)$}
```



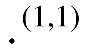
Here is the equivalence between **\uput**'s *refangle* abbreviations and **\Rput**'s *refpoint* abbreviations:

\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.

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)}
```

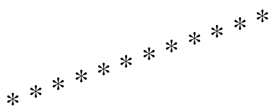
Driver notes: The rotation macros use **\pstVerb** and **\pstrotate**.

25 Repetition

The macro

\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: