9:05 a.m. June 11, 2009

To Anna, who helped me concentrate

PiScript—a drawing tool for mathematicians

by Bill Casselman

Preface

Much more than most mathematicians realize, producing good mathematical illustrations is a major part of good mathematical exposition. Computers have made this task totally different from what it used to be, but it is still not generally recognized as a simple procedure. I hope to change that with the program PiScript (πS) which this manual introduces.

πS is an interface to PostScript graphics, written in the well known programming language Python. It allows one to do basic programming in Python, but defines certain operators that interface very directly to the graphics commands in PostScript, which in turn produce PostScript files (and figures) as output. One of its best features is that inserting text into figures, even text produced by TEX, is straightforward.

But what's the point? There are already lots and lots of programs out there that will help you construct mathematical figures—`PSTricks`, `pictex`, `xfig`, PYX, `gnuplot`, and a host of similar programs of varying capabilities. Some of these also have a close relationship to PostScript, and some also allow TEX insertions. There are also as well the huge graphics components of Mathematica and Matlab and their open source simulacra such as `matplotlib` (which is the principal graphics component of SAGE). Why have I added yet another one to the collection?

None of the available programs makes it easy to construct simple figures and also more complicated ones of arbitrarily high quality. All of them have limited flexibility, compared to a direct use of PostScript. This is especially true of those that are ultimately based on TEX itself used as a graphics language. For one thing, in order to produce good mathematical illustrations it is necessary to do real programming, and the low-end tools do not make this easy or pleasant. In other words, a good graphics tool should be embedded in a real programming language. The high end ones can certainly produce plots and analyses of extraordinarily complicated data, but they fail as simple everyday tools, and cannot handle easily the more eccentric graphics tasks that mathematics often requires. And none allows the complete control of the graphics environment that PostScript provides. In addition, all but one of them have some trouble integrating text with graphics conveniently. The exception is PYX, but it is a very, very unwieldy tool otherwise,

One solution to this problem is the one I have myself used for many years—to program directly in PostScript. I have even taught PostScript as a graphics tool to undergraduates, in a course designed to help them understand the role of visual reasoning in mathematics. I have written the manual **Mathematical Illustrations** to go along with this project. But although I have managed to build an extensive library of programming tools to make it relatively easy for me to do good graphics work with PostScript, the complexity of my tools has eluded widespread adoption of my techniques by others. I won't list here all the problems one encounters when programming directly in PostScript, but there are many. I have in fact often thought how pleasant it would be to have some kind of object oriented graphics language with all of the good graphics output of PostScript but few of its other difficulties. I have had this idea in mind while constructing my own idiosyncratic tools, but when I first learned about Python, which was first called to my attention by William Stein, I realized that it would probably make my idea quite feasible.

The point of πS is that it makes my awkward work-arounds no longer necessary. It differs from many of the alternative graphics tools that I have mentioned in that it allows access to essentially *all* of the graphical features of `PostScript`, and there is thus no serious limitation on the quality of output. It differs from some of the more awkward graphics tools, those that embed graphics into TEX, in that it is itself embedded in Python, a convenient, elegant, and fully functional programming language. It differs from the direct use of PostScript in many ways. In particular, embedding of TEX text is easy, and one does not have to resort to opaque tricks to program effectively.

Another huge advantage of πS over PostScript is that you won't have to deal with the terrible, terrible error messages of PostScript. Well, not often, at any rate. Most of your errors will likely be made in Python, and errors in Python are handled admirably.

Compared to some graphics tools, πS is rather verbose. This is my own deliberate choice, and a matter of personal style—I prefer to offer the user relatively simple tools and let him build his own more complicated ones. One might think of πS as a kind of artist's tool rather than as a mathematical one. But then constructing a good mathematical illustration is in fact much like landscape painting. It is certainly more an art than a science. And the almost infinite flexibility at hand can make it seem as if those glorious days of kindergarten finger-painting can be relived. Mathematics becomes the toy it is already in our own minds.

This manual will cover only πS itself, and will say almost nothing about how to write a program in Python. I have written an appendix, however, with some brief advice on this. Documentation on PostScript itself will help you to understand the graphics model followed here. The book **Mathematical Illustrations** is an introduction to PostScript for those with some experience in mathematics. It has been published in tangible form by Cambridge University Press, and is also available at

`http://www.math.ubc.ca/~cass/graphics/manual`

As for the present document, there are four major parts:

1. Drawing in 2D
2. Drawing in 3D
3. Mathematics matters
4. Advice on illustrating mathematics

The last part is far from finished, and will be expanded in the future. In addition, there are several appendices:

A1. A brief introduction to Python
A2. Auxiliary programs
A3. Inserting PostScript figures into TEX files
A4. The make utility If you have followed
A5. Commands in the Vector class
A6. Index of commands

I would like to thank David Austin for helping me find errors in πS as it developed from a very small seed; William Stein for introducing me to Python; Christophe K. for helping me set up πS under Windows; and a few long-suffering students at the Berlin Mathematical School of TU-Berlin, and also in the graphics workshop at Simon Fraser University, for helping me chase out bugs.

Part 1. 2D drawing

There are some 3D capabilities in πS, and they will undoubtedly get better as time goes on. But it is mostly designed to make 2D pictures.

1. The graphics model

The graphics model of πS is essentially that of PostScript, and in fact its output is PostScript code that utilizes the PostScript graphics environment. PostScript is a complete programming language, but it was not designed primarily for human use. It was designed essentially as a sophisticated printer language, and even now nearly all of the world's PostScript originates in higher level graphics programs sending data to a PostScript-capable printer. It has what at first appears one very eccentric feature—like a few other languages (for example, FORTH) that were designed to be implemented efficiently on a physical machine, it is not compiled but fed in a straightforward way to the machine. It is designed to be executed as quickly as possible rather than to be written as conveniently as possible. This excludes the standard computer languages, in which—for example—the expression $2 + 7 * (3 + 5)$ can only be completely interpreted after all its subexpressions have been interpreted. Algebraic expressions are written in a context-free language and have to be **parsed** before interpretation. On the contrary, PostScript is expressed in RPN format, which allows commands to have immediate effect. (Reverse Polish Notation was invented for the most pure of reasons by Polish logicians early in the twentieth century.) This requires putting data before operators. For example, in PostScript adding x and y would be done as `x y add`. And the expression above would be evaluated by the sequence

```
2 7 3 5 add mul add
```

This is not so readable by humans, but to the computer it is very practical. Data is put on an *operator stack* and removed and operated on, as soon as possible by operators. For example, here is how the stack appears in the course of evaluating $2 + 7 * (3 + 5)$:

```
2
2   7
2   7   3
2   7   3   5   (add)
2   7   8   (mul)
2   56  (add)
58
```

No large expressions have to be put on hold until they have been completely read. As with the original HP calculators, in programming this way one has to keep mental track of the operator stack in order to do well with this scheme. This is one feature of the language that some never get used to, and indeed it occasionally causes even experts some perplexity. This ought not to be too surprising. Although there might very well be intelligent beings somewhere in the Universe whose mental processing is based on RPN processing, the human mind is surely based on the alternate paradigm of recursion and context-free grammar. But with the more conventional Python interface, that need not bother us.

The graphics model of PostScript is fairly simple. Firstr of all, there are two very different ways it produces graphics—one is by bit-mapped images, for example photographs, and the other is by constructing and manipulating paths. It is the second that we shall be concerned with (although in the future I'd like to see routines to do some bit-map manipulation). This is often called **vector** or **scaleable** graphics. The programs we shall write here draw paths. One can fill their interiors with color or merely stroke their outlines. The paths are constructed in a certain coordinate system, which the programmer can change as he or she goes along. When they are drawn, certain parameters (such as color) are applied. PostScript uses a stack in keeping track of the **graphics state**, which allows one to change graphics parameters but also to revert to previous values. In this it is like many other graphics languages.

The Wikipedia page

`http://en.wikipedia.org/wiki/PostScript`

is very instructive.

I now run through a brief description of the commands available in TS. Many of the commands have several alternative formats. In this preliminary manual, I have been rather brief. You should be able to figure out more by experimenting.

2. Getting started

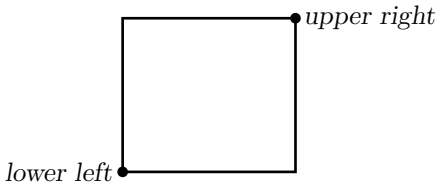
We might see some other options later on, but normally one begins a TS file by importing the Python module `PiModule`, located in the package `piscript`. Just about every TS program should thus start out with

```
from piscript.PiModule import *
```

In any TS program, you start with a call to the initialization function `init`, which sets the output file and the size of the figure (by specifying its dimensions). To preserve your sanity, your TS program files should always end with the extension `.py`, and the prefix of the `.py` file should match that of the PostScript output. So if the program file is `mydrawing.py`, the output file should be `mydrawing.eps`, referred to in the initialization simply as `"mydrawing"`.

- `init(filename, w, h)`
  `init(filename, llx, lly, urx, ury)`

The file name must be a string, surrounded by double quotes Normally, it will be the prefix of the output file. The full name of the output file will normally be prefix + `.eps`, but if you explicitly specify a filename with extension `.eps` or `.ps`, the output file will be the full file name. There is a subtle difference between the two possible extensions, but all you have to know is that if you are producing a PostScript file with several pages, the extension should be `.ps`. The corresponding PostScript figure will have bounding box `[0 0 w h]` (or `[llx, lly, urx, ury]`). I recall that the bounding box specifies the lower left and upper right corners of a figure.



The unit of length at startup is one Adobe point, or $1/72$ inch (or $2.54/72 = 1/28.35$ centimeters, since there are exactly 2.54 centimeters to one inch). This initialization also defines in PostScript the boundary of the figure. The numbers llx etc. can be floating point or integers, but they will be converted to the nearest integers because that's what the PostScript document structure specification demands.

- `beginpage()`
  `beginpage("noclip")`

Begins a new page. The important point about this is that we can output files of several pages, although usually there will be just one page produced. The pages must be isolated from each other—all changes in the graphics state should be entirely restricted to one page. At the start of every page, the unit of length is $1/72$ of an inch (one Adobe point), the coordinate grid is square, and the origin is at lower left. If the lower left corner of the bounding box is not the origin, you will probably want to follow this initialization by translating the origin to (llx, lly). Unless the argument 'noclip' is used, all subsequent drawing on this page is restricted to within the current bounding box. If 'noclip' is used, the figure will be allowed to overflow its nominal bounding box. Also, the default graphics state is saved on the graphics stack, and a new copy pushed above it.

• `endpage()`

Ends a page, restores the default graphics state so the next page can start out fresh. There *must* be matching `beginpage`/`endpage` pairs. The console will tell you as it is producing pages, and it will issue a warning if certain errors are encountered.

• `finish()`

All the rest of the time, πS is assembling a few large strings. At the very end of your file—and only once in your file—you should call `finish`. When this happens, these pieces are assembled and written to the output file, which is then closed. Forgetting to put this at the end of a πS program is fatal. A very common error in writing πS programs is to forget the parentheses in a command, for example writing `finish` instead of `finish()`. This will not cause an error in Python, because the name of a command without `()` is just seen as a pointer to the command. The command is silently ignored, and—worst of all—*there is no notice to this effect*. One sign that this has happened is that no `.eps` file is produced.

 *Be sure to finish every πS program with* `finish()`.

So the minimal πS program is something like

```
from piscript.PiModule import *

init("x", 100, 100)
beginpage()
endpage()
finish()
```

It opens a file called "x.eps", giving rise to a PostScript image of size very roughly 3.5 cm. square. But of course there is nothing to see there!

You can even have a file with two blank pages:

```
from piscript.PiModule import *

init("x.ps", 100, 100)
beginpage()
endpage()
beginpage()
endpage()
finish()
```

Next, we see how to construct paths and make them visible.

• `newpath()`

This starts a new path, destroying any previous one. Leaving this out when starting a new path is an extremely common error that will be passed over in silence by both PostScript and πS, but it will often lead to weird effects. What happens is that the new drawing commands just get added to those of the last path. Here, as with `finish()`, writing `newpath` instead of `newpath()` is a common error.

The real trouble with forgetting `newpath` is that often it will cause no harm at all. But when it does cause trouble it will often confuse you horribly. So I emphasize:

 *Be sure to start every new path with* `newpath()`.

• `moveto(x, y)`
`moveto([x, y])`
`moveto(P)`

This command puts the pen down at the position (x, y), making it the current point. **Every path must start with a moveto.** Here I follow a convention according to which P is a Vector, that is to say an instance of the Python class Vector to be discussed later on. The array and Vector forms of argument are useful (here, and also in other commands where they are acceptable) when feeding in points calculated by some other routine.

- `lineto(x, y)`
  `lineto([x, y])`
  `lineto(P)`

Adds to the current path the line from the current point to (x, y). Usually in drawing a path you want to start with a `moveto` and then continue it with a sequence of `linetos` to its end. But a path may have several components, each with its own initial `moveto`. Thus

```
moveto(-1,0)
lineto(1,0)
moveto(-1,1)
lineto(1,1)
```

constructs a pair of horizontal lines of length 2, one unit apart.

After you have constructed a path with `moveto` and `lineto` (or a few other drawing commands to be introduced later), you'll want to make it visible.

- `stroke()`
  `stroke(g)`
  `stroke(r, g, b)`
  `stroke([r, g, b])`

The commands so far described tell you how to construct a path, but they do not display it. There are two ways to do so. The first of these is `stroke`. It draws along the current path in gray scale g, or color (r, g, b). If no arguments are given, it strokes in the current color. At the start of every page, this colour is black. When a path is stroked, the coordinate system is the default, so that the default width is 1 point. But if you have scaled the line width or set it to something else, that change will take effect in the stroking *but in units of* 1 *point*.

The coordinates of a color should be in the range $[0, 1]$. The array form is convenient, since one can predefine colors: `red=[1,0,0]` etc. Higher is brighter, so black is $[0, 0, 0]$, white is $[1, 1, 1]$, and very light pink is $[1, 0.9, 0.9]$. I recall that grays are shades with equal RGB components. One could even define a whole collection of colors in some file somewhere and import them. One can also use the arrays to manipulate colors, for example by interpolating them or darkening them.

- `fill()`
  `fill(g)`
  `fill(r, g, b)`
  `fill([r, g, b])`

Similar to `stroke`, but fills the current path, implicitly first closing up each component of the path to its last `moveto`. I repeat: the commands `moveto` etc. construct a path, but do not render it visible. Only `stroke` and `fill` do that. So now here is a very simple program that actually draws something:

```
from piscript.PiModule import *

init("square", 100, 100)
beginpage()

newpath()
moveto(25,25)
lineto(75,25)
```

```
lineto(75,75)
lineto(25,75)
lineto(25,25)
fill(1,0,0)
stroke(0)

endpage()
finish()
```

This purports to draw a square, but we'll see later that it is flawed in a minor way. (See the discussion of `closepath`.) At any rate, what this produces is a PostScript file named `square.eps` that looks like this:

```
%!PS-Adobe-2.0 EPSF-3.0
%%Pages:  1
%%PageOrder:  Ascend
%%BoundingBox:  0 0 100 100
%%Creator:  PiScript Tue Jun  2 11:14:05 2009
%%BeginProcset:
/boundary { 0 0 moveto
100 0 rlineto 0 100 rlineto -100 0 rlineto
closepath } def

%%EndProcset

%%Page:  1 1
gsave
newpath
0 0 moveto
100 0 lineto
100 100 lineto
0 100 lineto
closepath
clip
newpath
25 25 moveto
75 25 lineto
75 75 lineto
25 75 lineto
25 25 lineto
gsave
1 0 0 setrgbcolor
fill
grestore
 gsave
0 setgray
[1.0e+00 0.0e+00 0.0e+00 1.0e+00 0.0e+00 0.0e+00 ] concat
stroke
  grestore
grestore
showpage
% ---------------------------------------
%%EndPage
%%Trailer
%%EOF
```

The PostScript file is a bit verbose, and rather difficult to read, but you should be able to track loosely what's going on without a lot of trouble. We'll see later what all those `gsaves` and `grestores` mean.

Incidentally, $\pi\mathcal{S}$ opens a temporary file on your system with the extension `.pys`, and if the program is interrupted it will probably leave one of these still around. You may remove it without hesitation.

3. More about drawing

Part of the graphics state in PostScript is the current path. Ultimately, it is to be incorporated in a figure by either filling its interior, drawing the path itself, or restricting the region affected by drawing to its interior. Paths are where the real action takes place—you might think of the part of a program that is actually constructing a path as its cockpit. Even text is ultimately just a collection of paths.

I have already introduced a few basic drawing commands. Here follow more:

- `rmoveto(dx, dy)`
  `rmoveto([dx, dy])`
  `rmoveto(V)`

Shifts the current point by the vector $[dx, dy]$, without adding anything visible to the current path. The V here is a Vector. I am just mathematician enough to distinguish between points (position) and vectors (displacement) in my notation, although in practice they are realized in the same Python class Vector. I'll say more about the distinction, which is important, in a later discussion about coordinate systems. For the moment, let me say that in the command `moveto(x,y)` the pair (x, y) is a point, because it represents position, but in `rmoveto(dx, dy)` the pair (dx, dy) represents a vector because it represent displacement relative to a position. As they tell you in physics class, a vector has direction and magnitude (but no more).

- `rlineto(dx, dy)`
  `rlineto([dx, dy])`
  `rlineto(V)`

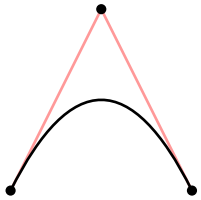
Adds a line segment to the current path, with end point relative to the current point. The `r` in `rmoveto` and `rlineto` stands for "relative".

- `quadto(x1, y1, x2, y2)`
  `quadto([x1, y1], [x2, y2])`
  `quadto(P1, P2)`
- `rquadto(dx1,dy1,dx2,dy2)`
  `rquadto([dx1, dy1], [dx2, dy2])`
  `rquadto(V1, V2)`

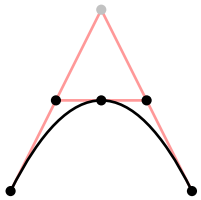
Curves can always be drawn as a sequence of small line segments, but there are two kinds of curves built in to $\pi\mathcal{S}$ that will look smoother. They are parametrically defined by quadratic and cubic parametrizations. The first command above adds to the current path a quadratic Bézier curve with control points (x_i, y_i). The second does the same, but the arguments are interpreted as coordinates relative to the current point. Quadratic Bézier curves are easy to imagine and to construct, since the control points have a simple geometric significance. The implicit start of this path segment is the current point and the last control point is the segment's end point, but here in addition the intermediate control point is the intersection of the tangent lines at the two endpoints. This makes quadratic Bézier curves the natural choice in many situations, and in particular when constructing contours.

```
newpath()
moveto(0,0)
quadto([10,20],[20,0])
stroke()
```

produces (tangent lines also drawn):

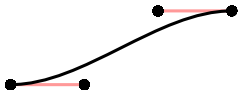


These and the cubic curves are drawn very efficiently by a computer, because of how they behave under subdivision. If a quadratic Bézier curve is divided in equal halves, each half is again a quadratic Bézier curve whose control points are simple to construct. The following figure illustrates what happens:



- `curveto(x1, y1, x2, y2, x3, y3)`
  `curveto([x1, y1], [x2, y2], [x3, y3])`
  `curveto(P1, P2, P3)`
- `rcurveto([dx1, dy1], [dx2, dy2], [dx3, dy3])`
  `rcurveto([dx1, dy1], [dx2, dy2], [dx3, dy3])`
  `rcurveto(V1, V2, V3)`

Adds to the current path the cubic Bézier curve with control points (x_i, y_i). See Chapter 6 of **Illustrations** for a discussion of Bézier curves. Roughly speaking, a Bézier segment begins at the current point, takes off towards $P_1 = (x_1, y_1)$, then winds up at $P_3 = (x_3, y_3)$ coming from the direction of $P_2 = (x_2, y_2)$. In these pictures, the control points are shown. In `rcurveto`, the arguments are relative to the current point.



There is one other very useful fact about Bézier curves that is useful in mathematical plotting. It is a relation between control points and calculus. Suppose we are given a parametrized curve $t \mapsto f(t)$ in the plane, and we wish to plot it. At the moment the only way we know how to do this is to plot it as a sequence of small—maybe very small—line segments. But occasionally this is a ridiculously difficult thing to do. If we are able to calculate the velocity $f'(t)$, we can use a smaller number of Bézier curves instead. If we want to plot the path between t and $t + \Delta t$ by a single Bézier curve, we know that the end points P_0 and P_3 are $f(t)$ and $f(t + \Delta t)$. It will follow from the discussion later on about Bernstein functions, since the coordinates of a Bézier curve are cubic Bernstein functions, that

$$P_1 = P_0 + (1/3)f'(t)\Delta t$$
$$P_2 = P_3 - (1/3)f'(t + \Delta t)\Delta t\,.$$

This is very useful for plotting trajectories of differential equations, and in particular integrals.

There is a third thing you can do with a path besides stroke or fill it.

- `clip()`

Clips subsequent graphics to the interior of the current path. In other words, it makes the current path the outline of a window through which we see what is drawn. The effects of this command can be controlled with `gsave/grestore`. The clipping path is part of the graphics state.

The only difference between the following pictures is the added three lines that clip to the box.

```
gsave()
newpath()
box(2,2)
clip()

newpath()
for i in range(N):
    moveto(-10, 10)
    lineto( 10, -10)
    translate(dx, dx)
stroke(1,0.8,0.8)
grestore()

newpath()
box(2,2)
stroke()
```

The clipping restriction should not be in place when the box is stroked.

- `closepath()`

Closes up the current path to the location of the last `moveto`. Even if the last point you draw to is the same as the first point you moved to, the path will not be in fact closed unless you use this operation—the beginning and end points will be treated differently from the other vertices of the path. The operation `closepath()` ensures that they are all considered democratically. The basic rule is simple:

If you really want to draw a closed path, use `closepath()`.

Also, `fill` automatically closes paths before filling. Thus we have the following three figures (with thickened line widths to exaggerate effects):

```
newpath()
moveto(0,0)
lineto(1,0)
lineto(1,1)
lineto(0,1)
fill(1,0.8,0.8)
stroke(0.6)
```

```
newpath()
moveto(0,0)
lineto(1,0)
lineto(1,1)
lineto(0,1)
lineto(0,0)
fill(1,0.8,0.8)
stroke(0.6)
```

```
newpath()
moveto(0,0)
lineto(1,0)
lineto(1,1)
lineto(0,1)
closepath()
fill(1,0.8,0.8)
stroke(0.6)
```

The rest of this section will be concerned with more complicated paths.

- `setdeg()`
- `setrad()`
- `todeg(x)`
- `torad(x)`

Angles in PostScript are measured in degrees. In πS one can set whether they are measured in degrees or radians with these commands. This affects how all angles are interpreted in πS operations that require angles. At the beginning, πS uses radians, and to tell the truth it is safer not to change. But it's awkward to have to write `math.pi/2` when you want $90°$. Furthermore, if you are going to use `math.pi` you must `import math` before you use it. Keep in mind that Python itself always uses radians internally, and this is not changed by either of these commands. The angle mode is not part of the PostScript graphics state and is not affected by `gsave/grestore`.

The command `todeg(x)` returns the angle in degrees that x represents in the current mode. For example, if the mode is 'degrees' then `todeg(90)` returns 90, but if it is 'radians' then 90 represents 90 radians, and it will return $90 \cdot 180/\pi$.

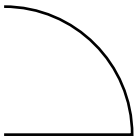
- `arc(x, y, r, A, B)`
  `arc([x,y], r, A, B)`
  `arc(P, r, A, B)`

Adds to the current path an arc in the positive direction from A to B, centred at (x, y) and of radius r. The angles A and B are interpreted according to the current angle mode.

There is some slightly unintuitive behaviour involved in `arc`*—if there is no current point, it starts with an implicit move to the beginning of the arc. If there is one, it adds a line from the current point to the beginning of the arc.*

Thus

```
newpath()
moveto(0,0)
arc(0,0,1,0,90)
stroke()
```



is different from

```
newpath()
arc(0,0,1,0,90)
stroke()
```

- `arcn(x, y, r, A, B)`
  `arcn([x,y], r, A, B)`
  `arcn(P, r, A, B)`

Goes in the negative direction.

- `circle(r)`
  `circle(x, y, r)`
  `circle([x,y], r)`
  `circle(P, r)`

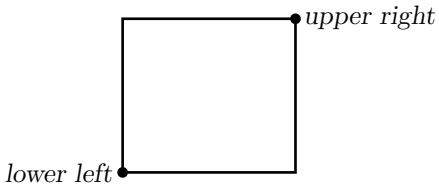
The same as a full circular arc, closed up. With just one argument, the center is at the origin.

- `box(w, h)`
  `box(x, y, w, h)`
  `box(P, V)`

The first adds a rectangle of width w and height h to the current path, with lower left corner at $(0, 0)$. The second has lower left corner at (x, y), the third at P.

- `bbox([llx,lly,urx,ury])`
  `bbox([llx,lly],[urx,ury])`
  `bbox(P1, P2)`
  `bbox(llx,lly,urx,ury)`

This is a *bounded box* with lower left corner at (llx, lly) and upper right corner at (urx, ury). This is especially convenient for use with TEX insertions.



- `graph(f, a, b, *args)`

Adds the graph of $y = f(x)$ from $x = a$ to $x = b$ to the current path. Here $f(x)$ is a Python function with one variable, and perhaps some extra parameters in the argument. By default, this command assembles 200 linear segments of uniform x-width. It also behaves like `arc`—if there exists a path already started it draws a line from the end of that path to the start of the graph. What `*args` means is that you can tack on to this an arbitrary number of extra parameters to be passed to the function. Here is the code for one version, which I include so you can see how simple it is:

```
def graph(self, f, a, b, *args):
    x = a
    N = 200
    dx = float(b-a)/N
    moveto(x, f(x, *args))
    for i in range(N):
        x += dx
        lineto(x, f(x, *args))
```

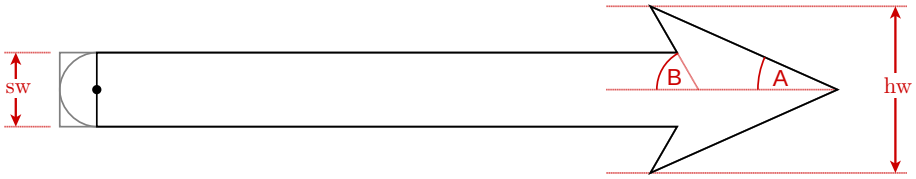
This is just about the simplest possible program you could use to draw a graph. Packages like Mathematica include far fancier routines to do this, and to handle all kinds of odd phenomena, but I prefer to roll my own. However, it is possible to do a smoother and in some sense more efficient construction of paths if you know how to calculate $f'(x)$ as well as $f(x)$. This involves Bézier curves. It has been mentioned already above, and is explained in more detail in Chapter 6 of **Mathematical Illustrations**.

4. Arrows

Arrows are an extremely, maybe surprisingly, common feature of mathematical diagrams. I have tried to make the package flexible. and easy to extend. First of all let me explain the most basic routines.

- `setarrowdims(sw, hw)`
  `setarrowdims(sw, hw, A, B)`

This sets the basic dimensions of all arrows. The dimension sw is that of the shaft width, hw the head width. The default for these is 1 and 3.6 in current length units, but any scale change by you will probably make you want to change them. The numbers A, B are certain angle parameters of the head. The default is $A = 24°$, $B = 60°$. What the tail of an arrow looks like is compatible with the current linecap style. In the following figure, all dimensions of the arrow are indicated, along with the different types of tails you get with distinct values of `linecap`.

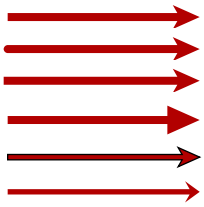


- `arrow(x,y)`
  `arrow([x,y])`
  `arrow(V)`
- `openarrow(x,y)`
  `openarrow([x,y])`
  `openarrow(V)`

The fitst constructs an arrow with its tail at the current origin and tip at (x, y). The second makes similarly an open arrow, as shown in a moment.

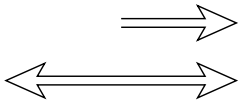
Here are some samples, with various values of sw and hw. In the last one, $A = 48°$, $B = 60°$.

Comment	Line cap
Flat tail	0
Rounded tail	1
Extended square tail	2
Wide head, 90° cut	0
Stroked and filled	0
Different A, B	0



Note that stroking the arrow thickens it noticeably, with the same construction command. It is usually a good idea to make line widths thinner when stroking an arrow.

The open arrows do not close up at the tail, so you can make double-ended arrows by fitting two together:



```
sw = 0.3
hw = 4*sw
setarrowdims(sw, hw)

gsave()
translate(0,-1)
newpath()
openarrow(4,0)
rotate(180)
openarrow(4,0)
stroke()
```

```
grestore()

gsave()
translate(0,1)
newpath()
openarrow(4,0)
stroke()
grestore()
```

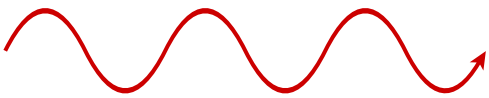
- `quadarrow(a)`

Builds an arrow using the argument a to assemble quadratic Bézier paths. The argument a is a sequence of pairs $[P, V]$, or a single array of such pairs, where P is a point and V an oriented direction (either can be an array or a Vector) the path takes from P. The sequence a must have at least two items in it, a beginning and an end. This is a very flexible structure.

In the current version, the tip of the arrow comes very close to hitting the last point of a.

```
a = [
    [ [-1,0], [1, 2] ],
    [ [0,0], [1,-2] ],
    [ [1,0], [1,2] ],
    [ [2,0], [1,-2] ],
    [ [3,0], [1,2] ],
    [ [4,0], [1,-2] ],
    [ [5,0], [1,2] ],
]

newpath()
quadarrow(a)
fill(1,0.8,0.8)
```

(in suitable units) produces



It would not be hard to write a procedure that turned any parametrized path into a sequence of Bézier quadratic curves, so one may contemplate constructing arrows that trace an arbitrary route.

- `texarrow(x)`

This creates an arrow that looks like those produced in TEX, commonly used in commutative diagrams. Here x is its nominal length. The shaft width is set by `setarrowdims`. The arrow goes from $(0, 0)$ to $(x, 0)$, so normally you'll apply coordinate changes when using this.

- `arcarrow(P, Q, r)`
- `arcnarrow(P, Q, r)`

Builds an arrow along an arc of radius r from P to Q. The direction is positive for `arc`, negative for `arcn`. Choosing a negative r makes the arc around the long way.

5. Graphics states I. The coordinate system

The graphics state at any moment records

(a) the current color
(b) the current coordinate system, which keeps track of the relation between the programmer's coordinates and those of some real physical space
(c) certain features of the lines it draws, such as dash pattern (default: solid), line width (default: $1/72$ of an inch), and the way lines end and join together
(d) the current region to which drawing is restricted (the **clipping path**)
(e) the current path being constructed

In the course of drawing something one might wish to change the graphics state only to go back to the old one after a while. To allow this, PostScript maintains an array of graphics states which one manipulates from time to time. It is a stack in the simplest sense—the basic operations are (a) adding on a new graphics state at the end of the array, or (b) removing the one currently at the end. When a new one is added, it starts up with a new copy of the current one, and changes in the graphics state are applied only to that copy. They do not affect previous graphics states. πS maintains a very minimal graphics state of its own, one that records the transformation from current user coordinates to the default system. For the rest, it just uses the one in PostScript, and many of the commands listed below are for that purpose.

• `center()`

Translates the origin of the coordinate system to the center of the bounding box. It is usually a good idea to do this after every `beginpage()`.

• `gsave()`

Pushes a new copy of the current graphics state onto the PostScript graphics stack.

• `grestore()`

Restores the previous graphics state.

It is a very good idea to sprinkle `gsave`/`grestore` *pairs liberally around in a program, encapsulating just about every graphics object you are dealing with.*

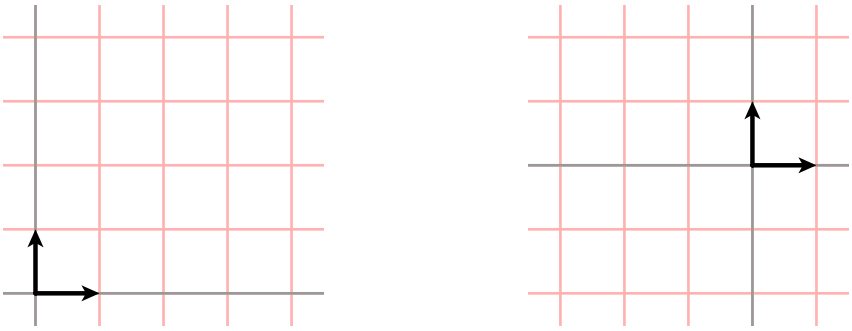
Then you can manipulate that object without affecting others. If there is a mismatch at the end of a page between the number of `gsave` and `grestore` operations on the page, πS will issue a warning by telling you the excess number of `gsaves` or `grestores`. *There is a serious error in your program if this happens, and you must locate it. Problems caused by it will almost certainly only magnify as your program develops.*

The rest of this section is concerned with coordinate changes. Understanding coordinate changes is the secret to efficient and flexible drawing in πS.

There are several commands with coordinates as arguments. These are the coordinates of points expressed in **user coordinates**, and in the process of drawing are translated immediately to default coordinates, and then in turn to hardware coordinates. The translation from one of these coordinate systems to another is done essentially in terms of a **coordinate frame**. This specifies the origin of the coordinate system and the unit vectors (displacements) along the x- and y-axes. **Coordinate changes change the frame.**

• `translate(x, y)`
`translate([x,y])`
`translate(V)`

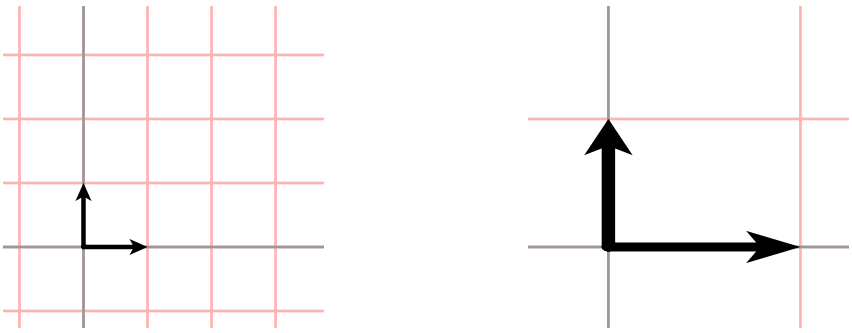
Translates the origin of the coordinate system by the vector $[x, y]$. This command is one of many $\pi\mathcal{S}$ commands that allows either a pair of coordinates or the corresponding vector as arguments. I have tried to make it possible to choose one or the other version in all relevant cases. I repeat: the effect of a coordinate change is to move the unit frame of the coordinate system to derive the frame of the new coordinate system. Thus `translate(3,2)` has this effect:



The point which used to be $(3, 2)$ is now the origin in the new coordinate system. The unit vectors along the axes are the same.

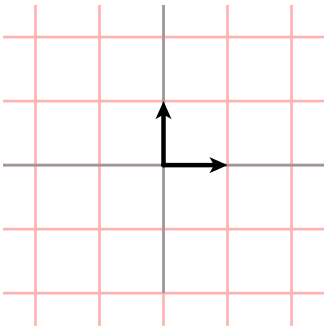
- `scale(s)`
  `scale(s, t)`
  `scale(unit)`

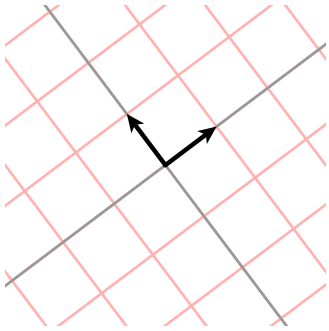
Scales both x and y by s, or x by s and y by t. the argument `unit` can be "cm", "in", or "mm", and if used in default coordinates will scale to that unit. Unlike in PostScript, *a scale change retains the current line width in absolute units*, and even if the x and y factors are different, *lines remain uniform in width*, whereas in PostScript lines in one direction might be thicker then in other directions. I thought a long time about this, because the PostScript model has definite aesthetic charm, but in truth I have never wanted non-uniform lines. In $\pi\mathcal{S}$ you can't have them. (Well, you can, but I won't tell you how.) The figures below illustrate that all dimensions *except line widths* are scaled. Again, I am showing the effect on a frame.



- `rotate(a)`
  `rotate(x, y, a)`
  `rotate([x,y], a)`
  `rotate(P, a)`

Rotates the current coordinate frame by angle a around the point (x, y) (or around the origin if (x, y) is not specified). In the figure, the angle is $37°$.





• `ltransform(a0, a1)`
`ltransform(a)` • `atransform(a0, a1)`
`atransform(a0, a1, a2)` `atransform(a)`

Applies a linear or affine transform apecified by the argument(s) to the current frame. The arguments for linear transform specify a 2×2 matrix, while those for the affine one specify in addition a shift vector. Each a_i is an array of two numbers. The interpretation is that a_0 and a_1 are the coordinates with respect to the current frame of the unit vectors of the new frame, and a_2 is the new origin. This is the most general coordinate transformation allowed. In case the arguent is a single array, it specifies either a_0, a_1 or a_0, a_1, a_2 in order.

Coordinate systems in PostScript and $\mathcal{TEX}$ are affine. A choice of coordinates is equivalent to specifying an affine coordinate frame with its origin at the coordinate origin and its side along the unit vectors from it. Coordinate changes move the frame in the obvious way. Thus this command applies the affine transformation corresponding to a to the current coordinate frame. This is discussed to some extent in Chapter 1 of **Mathematical Illustrations** and in more detail (much more detail) in Chapter 4, and I'll say more about this later on.

Let me summarize: the coordinate change operations are `scale`, `translate`, `rotate`, `ltransform`, and `atransform`. Actually, there is a mathematical theorem that asserts that all we absolutely need are translations, scalings, and linear rotations, but that would be awkward to depend on. It ought to give the idea, however, that you should be careful when combining coordinate changes, because you can in principal wind up with any affine coordinate transformation at all by combining simple ones. You can get weird effects.

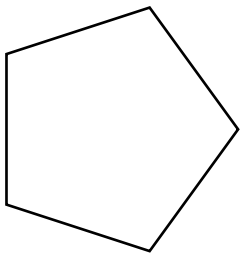
One may change the coordinate system while building a path, and *the interpretation of coordinates in commands* `moveto` *etc. is always in the current system*. Thus

```
moveto(1,0)
rotate(math.pi)
lineto(1,0)
```

produces a line between the points that were $(1, 0)$ and $(-1, 0)$ in the starting coordinate system. And

```
gsave()
newpath()
moveto(1,0)
for i in range(5):
    rotate(2*math.pi/5)
    lineto(1,0)
closepath()
stroke()
grestore()
```

draws this:



Of course boxes are interpreted in the current coordinate system. So

```
scale(2, 1)
newpath()
box(1,1)
stroke()
```

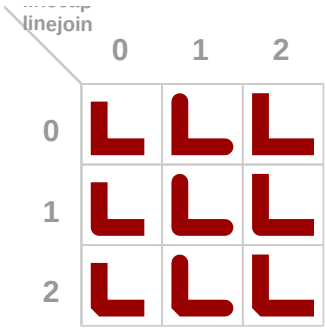
produces

6. Graphics states II. Other features

The most frequent manipulations of the graphics state involve coordinates, but there are other things involved, too.

- `setlinecap(n)`
- `setlinejoin(n)`

These determine what the ends of the lines, and the places where lines join, look like. Here $n = 0$, 1, or 2. Both 0 and 1 are useful—the first (default) makes square line ends and sharp line joins, whereas the second rounds things off. It is important in 3D drawing to set both of these to 1, and sometimes you want $n = 1$ for drawing arrows, but otherwise the default 0 is best. Here are the effects:

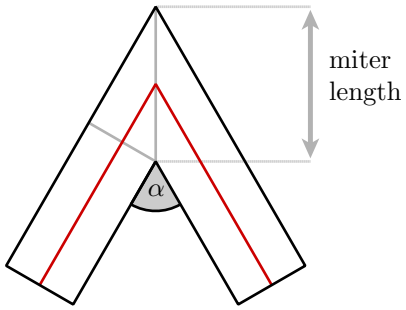


The style with `linejoin` set to 0 is called a **miter** style, that with 1 is **round**, and that with 2 is **beveled**.

- `setmiterlimit(x)`

This affects how lines are joined in the miter style. The number x must lie between 1 and ∞. To understand how this works, I have to explain a bit about the geometry of bevels and miters.

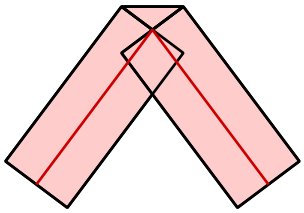
Suppose two lines join at angle α. The following diagram defines the **miter length**.



The diagram also shows that

$$\frac{\text{line width}}{\text{miter length}} = \sin(\alpha/2), \quad \text{miter length} = \frac{\text{line width}}{\sin(\alpha/2)}\,.$$

The following diagrams illustrate how beveling works:



The effect of `x setmiterlimit` is to set an angle α below which line joins are beveled. The angle α is such that

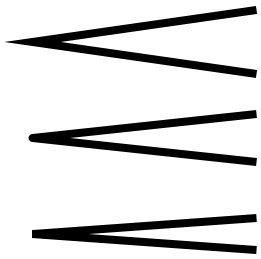
$$\frac{1}{\sin(\alpha/2)} = x, \quad \alpha = 2\arcsin(1/x)\,.$$

The larger x is, the smaller will be α. Some possible values of x and α:

x	angle α
10	11.47°
2	60°
$\sqrt{2} = 1.414\ldots$	90°
1	180°

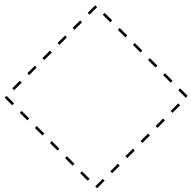
In the last case, all joins are beveled. The default value is $x = 10$.

For thin lines, the subtle points of line joins are not apparent except when the angle of intersection is quite small and the line join style is miter. In this case, they will certainly appear somewhat odd, and to avoid it you will want to set the style to round or bevel. Bad effects are particularly prominent in 3D drawing, so when doing 3D drawing I myself almost always set the line join style to 1 (round).



- `setdash(a, b)`

Sets current stroke pattern to a dashed line. Here a is an array of lengths setting the on/off pattern, b is the initial offset. In the following figure, the side of the square is 1, $a = [4, 4]$, $b = 0$.



The dimensions of the arguments are points, since `stroke` is called in default coordinates.

- `scalelinewidth(c)`

Multiplies the current line width by c. The line width is part of the graphics state.

- `setlinewidth(c)`

Sets the line width. *The unit in which line width is measured is always points.* It is equal to 1 point unless changed with one of these two commands.

- `currentlinewidth()`

Returns the current line width. This may be used to match the widths of arrows and lines when in the default coordinate system.

- `setcolor(r, g, b)`
  `setcolor([r, g, b])`
  `setcolor(g)`

Sets the RGB (red, green, blue) components of the current color or sets the current color to a shade of gray g in $[0, 1]$. Gray g is equivalent to $[g, g, g]$.

```
red = [1,0,0]
setcolor(red)
```

I should mention that the clipping path is also part of the graphics state. At the start, there is none, but after `beginpage()` it is the bounding box.

Part 2. Putting text in figures

There are two quite different ways to put text in πS figures. One allows you to embed TEX into them. The great virtue of this is that you can use TEX to handle the formatting of the text, especially of mathematics. The other is to use the enhanced graphics capabilities of PostScript to place non-mathematical text in possibly more creative ways. Eventually I'd like to merge these two streams so as to enhance both.

7. TEX Text

The basic object for placing TEX labels into a figure is a `TexInsert` object. It has an existence independent of where it is to be placed, and includes in its data a number of dimensions and locations as well as the text to be placed.

• `texinsert(texstring)`

Returns a TexInsert object. The default origin of the insert is at the origin of the first character in the TEX string.

There is one important thing to watch out for here:

In writing TEX macros in a string that is an argument for a TEX insert, \ *should always be written as* \\.

This is because the escape \ is used in Python to access special characters. Thus `\n`x in a Python string means 'linefeed'. Thus to put π in a TEX insert one should write `$\\pi$`. It is not in fact always necessary, and not even in this case. There are only special situations in which the \ gets lost. But putting in \\ never fails.

In making TEX inserts πS will similarly make temporary files that begin with `tmp` and have as prefix `.dvi` or `.log` or `.aux`—the usual garbage files that TEX creates. Again, if production is interrupted these may be hanging around, and should be deleted.

• `place(t)`

Places the TexInsert t with its origin at the current coordinate origin. I repeat:

The command `place(t)` *puts the TexInsert* t *with its origin at the current coordinate system origin.*

This may not not quite what is expected. Many people think the behaviour should be as with `show`, where the text is placed at the current point. However, placing a TexInsert inserts a full external PostScript file, and is more like `importEPS` than `show`. At any rate, you will usually preface `place` with a translation, and encapsulate it with `gsave/grestore`.

A TexInsert has several data attached to it. One is the output from turning the TEX string into an encapsulated (i. e. stand-alone) PostScript file, which is to be inserted wholesale into a figure's PostScript file. It also has a `bbox`, `width`, `height`, `depth`, `origin`, `center`, and an array of locations stored in an array `mark`. The height is the difference in height between the top of the bounding box and the origin, the depth is the difference between the origin and the bottom. The origin is by default the left bounding point of the first character set. This works well to allow alignment of different texts. The bounding box can be used, among other things, to blank out space behind a TEX insert.

Let's look at some more examples of TEX placement.

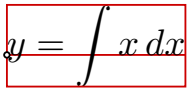
```
t = texinsert("abc")
gsave()
translate(100,100)
newpath()
bbox(t.bbox)
stroke()
```

```
place(t)
grestore()
```

draws the outline of the bounding box of the text around the text itself, which is placed at $(100, 100)$.

```
t = texinsert("$$y = \int x\, dx$$")
translate(-0.5*t.width, 0)
newpath()
moveto(0, 0)
rlineto(t.width, 0)
stroke(0.8,0,0)

place(t)
newpath()
bbox(t.bbox)
stroke(0.8,0,0)

newpath()
circle(0,0,0.75)
fill(1)
stroke()
```

produces



The size of the placement will be in current units, so if the natural size of the TEX font is 10 points and you have scaled by 10 the letters will appear at roughly 100 points in height. So you might want to temporarily scale down before placement. My usual practice in making TEX placements is to use `revert`, which is explained later.

A TEX insert has a few more useful commands.

- `(TexInsert).setorigin([x, y])`
  `(TexInsert).setorigin(x,y)`
  `(TexInsert).setorigin(V)`

This sets the origin of the TEX insert at the point (x, y) in its current coordinate system—i. e. it shifts the origin by (x, y). Thus whereas

```
t = texinsert("abc")
place(t)
```

will place the TEX insertion with the origin of its 'a' at the origin of the current coordinate system,

```
t = texinsert("abc")
t.setorigin(t.center)
place(t)
```

will place its center at the current origin, and

```
t = texinsert("a\ mark bc")
t.setorigin(t.mark[1])
place(t)
```

will place the origin of the character of the 'b' at the current origin. In all cases, the TEX insertion is considered to be in its own coordinate system. Experiment will let you see how this works.

8. The TEX environment

The way `texinsert` executes depends on the πS TEX environment, which consists if three items—the TEX command, the TEX prefix, and the TEX postfix. In this situation, a temporary file is assembled from the prefix, your TEX string, and the postfix, and then the command is run on that file. These are read from a configuration file.

The basic πS package contains the configuration file `TexConfig.py` in which these are set, and this is the default. If you want to redefine them globally, you should rewrite this file before installing πS. But there are other options. First of all, you may load other TEX configuration files from the global πS directory. You may also place such files in your own `PYTHONCONFIGDIR/configs` directory, or load them from your current working directory. These are the places where πS will look when importing modules.

The default command is "latex", the default LATEX prefix is

```
\documentclass[12pt]{article}
\pagestyle{empty}
\input amssym.def
\newcommand\color[1]{\special{color{#1}}}
\newcommand\uncolor{\special{uncolor}}
\newcommand\lmark{\special{mark}}
\begin{document}
```

and its postfix is

```
\end{document}
```

This means that when, for example, you put in your program `place(texinsert("$y= x^2$))` a TEX file like this is produced:

```
\documentclass[12pt]{article}
\pagestyle{empty}
\input amssym.def
\newcommand\color[1]{\special{color{#1}}}
\newcommand\uncolor{\special{uncolor}}
\newcommand\lmark{\special{mark}}
\begin{document}
$y = x^{2}$
\enddocument
```

and then LATEX is run on it, eventually to produce an `.eps` file to be imported into your own `.eps` file.

If you are happy with these choices, you can skip to some other section.

But you are not at all restricted to this standard option. The `TexConfig.py` that I myself use is a plain TEX environment, with prefix

```
\input amssym.def
\def\color#1{\special{color{#1}}}
\def\uncolor{\special{uncolor}}
\def\mark{\special{mark}}
\nopagenumbers
```

postfix \bye, and command "tex > /dev/null 2> /dev/null". This command suppresses all output from TEX. This file is installed as TexConfig.py in my directory PYTHONCONFIGDIR/configs, which is searched for a file TexConfig.py before the main TS directory is. The file I use is found as PlainTexConfig.py in the TS distribution. You can get other environments, however, in several ways.

- settexprefix(s)
- settexpostfix(s)
- settexcommand(s)
- settexenv("plain")
  settexenv("latex")
  settexenv(texconfigfile)
  settexenv(prefix, postfix, command)

In the first three, s is a string, and so are the arguments in the last one.

One option for texenv is the prefix of a TEX configuration file, more or less similar to the global TexConfig.py. It can refer to a file in either the current working directory, the directory PYTHONCONFIGDIR/configs, or the principal TS directory. These are searched in that order. The argument "plain" is equivalent to "PlainTexConfig", "latex" to "LaTexConfig". This option makes things much more flexible, because it can give you complete control over how TEX is executed. Here, for example, is a file PSTexConfig.py that I use, calling for it with settexenv("PSTexConfig").

```
def defaultTexEnv(s):
    p = "\\input amssym.def\n"
    p += "\\def\\color#1{\\special{color{#1}}}\n"
    p += "\\def\\uncolor{\\special{uncolor}}\n"
    p += "\\def\\mark{\\special{mark}}\n"
    p += "\\nopagenumbers\n"

    f = open("psfont.defs", "r")
    p += f.read()
    f.close()

    q = "\n%\n\\bye\n"
    # c = "tex"             # use this for Windows
    c = "tex >& /dev/null"     # this suppresses TEX error messages
    return piscript.Tex.TexEnv(p, q, c)

def getTexEnv():
    return defaultTexEnv("plain")
```

Here, psfont.defs is a file defining TEX macros that allow me to use Adobe fonts in TEX, roughly like this:

```
\font\tenrm=pplr at 9.5pt
\font\tensl=pplro at 9.5pt
...
\rm
```

9. PostScript Text

There are also available a small number of methods that will place ordinary PostScript strings in your figure.

- `setfont(f, s)`

Sets the current font to be f, with nominal size s. There are a small number of fonts always available in PostScript, and in practice one should normally use only those. (Later versions of TikZ will make this advice obsolete.) They are

`/Times-Roman`
`/Times-Italic`
`/Times-Bold`
`/Times-BoldItalic`

`/Helvetica`
`/Helvetica-Oblique`
`/Helvetica-Bold`
`/Helvetica-BoldOblique`

Most sites also have available in addition the fonts

`/Bookman-Demi`
`/Bookman-DemiItalic`
`/Bookman-Light`
`/Bookman-LightItalic`

`/AvantGarde-Book`
`/AvantGarde-BookOblique`
`/AvantGarde-Demi`
`/AvantGarde-DemiOblique`

`/Palatino-Roman`
`/Palatino-Italic`
`/Palatino-Bold`
`/Palatino-BoldItalic`

`/NewCenturySchlbk-Roman`
`/NewCenturySchlbk-Italic`
`/NewCenturySchlbk-Bold`
`/NewCenturySchlbk-BoldItalic`

`/ZapfChancery-MediumItalic`
`/ZapfDingbats`

If you want to use other fonts, you should first load them into your program. I'll say more about that possibility later on.

- `shift(r, s)`

This is a bit technical. It translates horizontally by r times the width of the string s if it were placed in the figure. For example, to center a string at a point you would apply this command with $r = -0.5$. The reason for this awkward format is that TikZ knows nothing about the geometric dimensions of a string when it is placed in a figure, so a PostScript sequence has to be invoked to use this information. This command involves a coordinate change, and is best encapsulated with `gsave/grestore`. *This must be used only after the font has been set*, or the shift will be applied to the empty font.

- `show(s)`

Displays the string s at the current point in the current font. Normally this is immediately preceded by a `moveto`. Thus

```
setfont("/Helvetica-Bold", 12)
moveto(100,100)
s = "Hello!"
shift(-0.5, s)
show(s)
```

places the string "Hello" in Helvetica-Bold with nominal size 12 units, centred at the point $(100, 100)$.

• `PScharpath(s)`

adds the outline of the string `s` to your current path. Thus

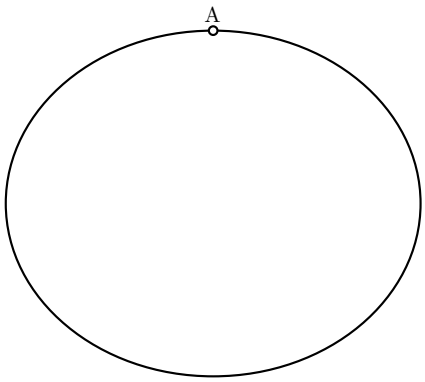
```
setfont("/Helvetica-Bold", 36)
s = "abc"
shift(-0.5,s)
newpath()
moveto(0,0)
PScharpath(s)
fill(1)
stroke()
```

produces

10. Reversion to default coordinates

There is one very useful feature I haven't mentioned yet, because although it is very useful some find it difficult to use.

You will normally not want some parts of a figure to be affected in certain ways by coordinate changes. This is particularly true of text in your figures. For example, suppose you want to put a label 'A' at a certain point of a diagram. You will probably want to have that letter at the same size as your encompassing text. Furthermore, if you shift the point in some later version, you will want the vertex to move with it. So the size of the label should be specified in the default coordinate system, but its placement should be in terms of the current one. For another example, you want to draw an ellipse, which you then label by an 'E' at its center. You draw the ellipse by non-uniform scaling, but you certainly don't want the text to be scaled non-uniformly. In either of these examples, you might want to locate your point visually by drawing a small circle at it. But the size of the circle should be expressed in terms of the default coordinate system. *How do you achieve these effects?*



The ellipse here is drawn as circle with top at $(0, 1)$in user coordinates; the label 'A' is produced by TEX in Computer Modern Roman at the default font size of 10 points.

- `revert()`

This reverts back to the default coordinate system, and returns a copy of the **graphics state** current at the time it is called. You can use it, say, to transform vectors. The current graphics state specifies at any moment the transformation from the current coordinate system to the original one, as well as a few other things such as the current line width. More precisely, it returns an instance of a class `GraphicsState` that specifies these. You will not normally need to know anything about the internal structure of this class, but only that it has associated to it a number of procedures that you can use.

- `(GraphicsState).transform(x,y)`
  `(GraphicsState).transform([x, y])`

This returns the transform of (x, y) into default coordinates by the coordinate transformation of the graphics state.

- `(GraphicsState).rtransform(dx,dy)`
  `(GraphicsState).rtransform([dx, dy])`

This returns the transform of the *vector* $[dx, dy]$ by the coordinate transformation of the graphics state. That is to say it returns the transform of the point (x, y) minus that of $(0, 0)$. Useful for making arrows when `revert` has been called. (The 'r' stands for 'relative'.)

- `(GraphicsState).itransform(x,y)`
  `(GraphicsState).itransform([x, y])`

This returns the transform of (x, y) by the inverse of the graphics state.

For example, here is the code that produced the figure above (I give almost the entire file):

```
# we start out in the default coordinate system at level 0
beginpage()   # saves the default graphics state, adds a copy at level 1
center()      # the graphics state at level 1 now has origin at center
scale(96, 80) # and a non-uniform scaling

newpath()
circle(1)     # the non-uniformity means this circle is really an ellipse
stroke()

  # we want to place a label "A" at the top of the ellipse
gsave()       # now at graphics level 2
```

```
gs = revert() # gs is a copy of graphics state at level 2
# and we are now back to the default coordinate system

v = gs.transform(0, 1) # v is the vector [1, 0] in original coordinate system
  # now expressed in default coordinates
translate(v)  # translate the origin to that point

newpath()
circle(2)     # this is a true circle of radius 2 points
fill(1)
stroke()

translate(-4, 4)  # still in units of points
t = texinsert("A")
place(t)
grestore()    # now at graphics level 1 again

endpage()     # now at level 0 again
finish()
```

The code after `gsave()` reverts temporarily to the default coordinate system, but has access to the current one as well.

The rest of this section looks under the hood.

- `cgs()`

Returns a copy of the current graphics state.

The next few commands are all applied to an instance of the class `GraphicsState`.

- `(GraphicsState).tm()`

This returns the transformation matrix of the graphics state to which it is applied. This specifies the transformation from the coordinate system associated to that graphics state to default coordinates. It is returned as an array of three vectors

$$[[t_{00},\ t_{01}],\\ [t_{10},\ t_{11}],\\ [t_{20},\ t_{21}]]$$

representing the affine transformation

$$[x,\ y] \longmapsto [x,\ y] \begin{bmatrix} t_{00} & t_{01} \\ t_{10} & t_{11} \end{bmatrix} + [t_{20},\ t_{21}] = [t_{00}x + t_{10}y + t_{20},\ t_{01}x + t_{11}y + t_{21}]\,.$$

You should not normally have to refer to the internal structure of the matrix.

- `(GraphicsState).inversetm()`

Returns the inverse of the transform matrix associated to the graphics state to which it is applied.

the command `revert` is just a combination of these commands. Here is its definition:

```
def revert():
    gs = cgs()
    t = gs.inversetm()
    atransform(t)
    return(gs)
```

11. More about TEX inserts

The command `texinsert` returns an instance of the Python class `TexInsert`. The basic parameter is a single string. A TEX file is made up from this string, preceded by a prefix, and followed by a postfix. As I have already mentioned, the default prefix is

```
\documentclass[12pt]{article}
\pagestyle{empty}
\input amssym.def
\newcommand\color[1]{\special{color{#1}}}
\newcommand\uncolor{\special{uncolor}}
\newcommand\lmark{\special{mark}}
\begin{document}
```

A TEX file is created from a TEX insertion, and the command `tex` applied to it, creating a `.dvi` file which is read into a program that creates an `.eps` file from it. The important thing to note is that certain TEX specials are allowed here, which are interpreted by the `.dvi` reader in $\pi\mathcal{S}$. The special `\mark` places the current point's coordinates in a list attached to the `TexInsert` structure, and the pair `\color/\uncolor` allow temporary color changes within the TeX insert. **These must occur in matched pairs.** Here is a sample use of the color macros:

```
t = texinsert("a\\color{1 0 0}b\\uncolor c")
```

I might have implemented the standard colour macros in `colordvi.tex`, but I decided that the standard colours provided there weren't interesting enough, and that the cmyk (rather than rgb) conventions it follows weren't intuitive enough.

I repeat: keep in mind that the character `\\` in a Python string must always be written as `\\ \\` .

- `font(fn)`

This returns a structure that I call a $\pi\mathcal{S}$ **font** whose `.pfb` file is `fn`. For example,

```
font("/usr/share/texmf/fonts/type1/bluesky/cm/cmr10.pfb")
```

returns the $\pi\mathcal{S}$ font associated to `cmr10`. There is a lot of structure hidden inside one of these fonts, but at the moment the only useful thing to do with one is to get character paths:

- `charpath(f, n)`

This returns the path of the character with index n in font `f`. I'll say later what this means, and how to use the path.

12. Paths

For some purposes, it is useful to build and store a path without drawing it.

- `path()`

This returns an empty `Path`. A real path is constructed by the usual operations `moveto` etc. The actual path is an array of integers and floating point numbers, basically an array of sequences amounting to command plus arguments describing a 2D path. The commands are (i) 0, meaning `moveto`; (ii) 1, meaning `lineto`; (iii) 2, meaning `curveto`; (iv) 3, meaning `closepath`. Each command is followed by the appropriate number of arguments, for example 6 for `curveto`, and none for `closepath`. Every one of our paths in 2D can be expressed in such a fashion, and in particular the paths of characters in TEX fonts are naturally expressed in such a way. The commands accessible to a path are essentially the same used in constructing the usual paths: `moveto`, `lineto`, `rmoveto`, . . . A Path even has its own graphics stack, so you can make coordinate changes while constructing it.

The point of this is that often one wants to draw the transformation of a path, and it is straightforawrd to transform paths given as arrays of the kind described here. We shall see later an example of transforming a 2D path into 3D, but it is easy to think up other examples of this technique.

Essentially, all compound objects should be planned to be paths. For example, Arrows are in this category. PiScript introuces a Python class `Arrow` and this is the class through which arrows of all kind are constructed from a head, a shaft, and a tail. The class `Arrow` possesses a procedure called `mkpath`, and this can be called with either an instance of the class `PiScript` or of the class `Path` as argument. The call `a.mkpath(p)` (where `a` is an Arrow) then either constructs the path in PostScript or as a Path to be used later. Other compound objects to be dealt with in this way might be arcs. Characters of certain fonts already have this feature, and in the future I expect to be able to render an entire `.dvi` file as a Path, which would allow you to deal with TEX output very flexibly.

13. More about PostScript font usage

I have listed the standard PostScript fonts available wherever a PostScript interpreter can be found, but it is not necessary to use only these. If you use any other fonts, however, you must explicitly load those fonts into your program. You can do this at any point, but you will probably remain a bit more sane if you do this right at the beginning, just after `init`.

You can load the fonts with the `importPS` command. But you must load them in `.pfa` format. What does this mean? On most systems that have PostScript fonts, they are stored in a compressed `.pfb` format, where the `b` in `.pfb` stands for *binary*. These are not legitimate PostScript files, and if you load them into a PostScript file directly they will result in a mess. You must first convert the `.pfb` to a `.pfa` file, where the `a` in `.pfa` stands for *ascii*. These are legitimate PostScript files. There is a common utility program called `pfbtopfa` that will carry out the conversion. Thus, if you have the file `cmr10.pfb` (a standard TEX font) in your current directory

```
pfbtopfa cmr10.pfb
```

will create the file `cmr10.pfa`. After that, you can do this:

```
from piscript.PiModule import *

init("fonts", 64,36)
importPS("cmr10.pfa")

center()
translate(0,-16)
setfont("/CMR10", 36)
s = "abc"
shift(-0.5,s)
moveto(0,0)
show(s)
endpage()
finish()
```

to produce:

Note that the name of the file is `cmr10` but the name of the font in the file is `CMR10`. In general, a `.pfa` file contains its name towards the beginning of the file, often right on the first line. Thus the first line of `cmr10.pfa` is

```
%!PS-AdobeFont-1.1:  CMR10 1.00B
```

Don't forget the / in the name you use!

You will likely see some problems in using TEX fonts in this way, however, because the **encoding** of these fonts is not the standard ascii encoding. In particular, the space is not in the usual place, so if you put in `(a b c) show` you'll get something weird. Fonts other than TEX fonts are usually better behaved, and there are many free fonts available on the 'Net. The following program will let you look at a whole segment of a font (in this case, `cmr10` again):

```
from piscript.PiModule import *

init("cmr", 200,100)
importPS("cmr10.pfa")

def octal(n):
    if n == 0:
        return "\\000"
    elif n < 8:
        return "\\00" + str(n)
    elif n < 64:
        return "\\0" + str(n/8) + str(n%8)
    else:
        s = "\\"
        s += str(n/64)
        n = n % 64
        s += str(n/8)
        n = n % 8
        s += str(n)
        return s

beginpage()
cw = 10
ch = 10
gsave()
translate(0.5*width(),8*ch)
translate(-cw*8,0)
setfont("/CMR10", 10)
for i in range(128):
    m = i/16
    n = i % 16
    moveto(n*cw,-m*ch)
    show(octal(i))
grestore()
endpage()
finish()
```

The only tricky part of this is realizing that you have to convert a character index to octal form to display that character in a PostScript string. Thus to display the character whose index is $65 = 8^2 + 1$ you put `(\101) show` in the PostScript file. One thing this program ought to remind you of is that you should always be willing to define your own subroutines. In fact, for sophisticated graphics tasks this will be a necessity. Anyway, here is the product:

There are lots of free fonts on the 'Net, but few that are really practical, and few that are of high quality. I myself am rather fond of

```
http://manfred-klein.ina-mar.com/
```

which is a curious medley of all kinds of stuff.

Part 3. 3D drawing

There is a simple library of 3D operations in the PiModule package. These can be accessed by calling `init3d` instead of `init`, with the same arguments.

14. Simple 3D drawing

The 3D environment is more complicated than that in 2D. The eye is assigned a fixed location along the z axis, looking down the negative z-axis. The images one actually sees are those you get by projecting onto the plane $z = 0$. There are operations for drawing lines in 3D, but also some more complicated ones for seeing surfaces, with some ambient lighting. In designing this package, I was not concerned with realistic effects, but providing just enough features to help the human eye interpret 3D images. As in 2D, coordinate changes move the base frame, which starts out as the standard rectangular coordinate system.

In drawing 3D objects, it is usually best to set line joins to be 1, or weird things will appear.

The new commands are:

- `gsave3d()`
- `grestore3d()`

There is a stack that keeps track of the 3D graphics state in the same way the one in 2D does. At the moment it stores only of the coordinate system. It is completely independent of the 2D graphics state, and is not changed with new pages.

- `seteye(e)`

Here $e = [x, y, z, w]$ is a 4D array (or, as in all these commands, a Vector). The coordinates are interpreted as homogeneous, which means that scaling them by a positive scalar doesn't change the interpretation. (I'll say something about the mathematics of homogeneous coordinates later on.) At the moment x and y must be 0, and both w and z ought to be positive unless you want to see weird things. If $w = 0$ the eye is set at infinity, otherwise at the 3D point $(0, 0, z/w)$. Internally πS3d works entirely with homogeneous 4D coordinates, because it makes computations involving perspective very simple.

- `scale3d(a,b,c)`

Scales x, y, z.

- `rotate3d(a, A)`

Rotates by angle A around axis $a = [x, y, z]$. Angles are interpreted as degrees or radians depending on the angle mode.

- `translate3d(x, y, z)`
  `translate3d([x,y,z])`
  `translate3d(V)`

Translates the coordinate frame.

- `moveto3d(x,y,z)`
  `moveto3d([x,y,z])`
  `moveto3d(P)`

Starts a path.

- `rmoveto3d(dx,dy,dz)`
  `rmoveto3d([dx,dy,dz])`
  `rmoveto3d(V)`

- `lineto3d(x,y,z)`
  `lineto3d([x,y,z])`
  `lineto3d(P)`

- `curveto3d(P1,P2,P3)`

Here the arguments are 3D points, assumed to lie in a plane.

- `rlineto3d(dx,dy,dz)`
  `rlineto3d([dx,dy,dz])`
  `rlineto3d(V)`

- `closepath3d()`

Here is an example. It shows on successive pages a rotating square frame, with the eye set at $(0, 0, 1)$:

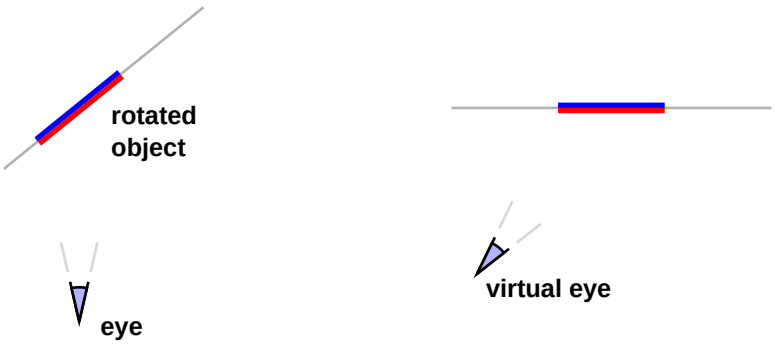
```
from piscript.PiModule import *

init3d("rotatingframe", 250, 150)
import math

seteye([0,0,10,1])
for i in range(36):
    beginpage()
    center()
    scale(100)
    translate(0,-0.5)
    newpath()
    moveto3d(0,0,0)
    lineto3d(1,0,0)
    lineto3d(1,1,0)
    lineto3d(0,1,0)
    closepath3d()
    stroke(0)
    endpage()
    rotate3d([0,1,0], math.pi/18)
finish()
```

15. Less simple 3D drawing

In real life, what we see is affected by ambient light. Opaque objects hide other objects. A simple form of these phenomena are taken into account here.

- `geteye()`

This returns the **virtual eye**, which is the eye placed where it would be if the coordinate system were inverted. Thus if the coordinate system rotates around an axis a by angle A, the virtual eye rotates around a by $-A$. The virtual eye is used to check visibility and lighting. See Chapter 10 of **Illustrations**. What happens is illustrated in these figures:



In the following program, as the square rotates one way, the virtual eye rotates the other. When the virtual eye is on one side of the *original* square, the real eye, which is fixed, is on the same side of the rotated square. On one side it sees red, and on the other blue.

```
from piscript.PiModule import *

init3d("rotatingsquare", 250, 150)
import math

seteye([0,0,10,1])
for i in range(36):
    beginpage()
    center()
    scale(100)
    translate(0,-0.5)
    e = geteye()
    newpath()
    moveto3d(0,0,0)
    lineto3d(1,0,0)
    lineto3d(1,1,0)
    lineto3d(0,1,0)
    closepath3d()
    if e[2] > 0:
        fill(1,0,0)
    else:
        fill(0,0,1)
    stroke(0)
    endpage()
    rotate3d([0,1,0], math.pi/18)
finish()
```

- `setlight(L)`

The vector L is also 4D, with last coordinate required to be 0. Sets the direction from which light comes. Internally, the light is a unit vector.

- `getlight()`

Returns the virtual light source.

In the next example, the square is shaded very crudely according to where the light is located with respect to the normal vector of the surface that is visible.

```
from piscript.PiModule import *

init3d("litsquare", 250, 150)
import math

seteye([0,0,10,1])
setlight([-1,1,0.5,0])
for i in range(36):
    beginpage()
    center()
    scale(100)
    translate(0,-0.5)
    e = geteye()
    L = getlight()
    newpath()
    moveto3d(0,0,0)
    lineto3d(1,0,0)
    lineto3d(1,1,0)
    lineto3d(0,1,0)
    closepath3d()
    if e[2] > 0:
        s = (1+L[2])*0.5 + 0.5
        fill(s,0,0)
    else:
        s = (1-L[2])*0.5 + 0.5
        fill(0,0,s)
    stroke(0)
    endpage()
    rotate3d([0,1,0], math.pi/18)
finish()
```

16. More about surfaces

In $\pi\mathcal{S}$ as in most computer graphics programs, a surface is an assembly of flat polygons. Maybe a really huge number of small polygons, in an attempt to simulate a smooth surface such as a sphere, but still ultimately an assembly of polygons. After all, even in nature the apparent smoothness of surfaces is an illusion.

How we see surfaces is a function both of the qualities of the surface itself and the ambient light. $\pi\mathcal{S}$ is not interested in providing realistic illusions, but only in offering enough clues to the human eye so that it understands roughly what it is seeing; it is lucky (for $\pi\mathcal{S}$) that the human eye is easily fooled, and in fact cooperates happily in being fooled.

$\pi\mathcal{S}$ offers two kinds of surfaces, polyhedral and smooth. A polyhedral one is just an assembly of its faces, where each face is constructed from a flat 3D polygon and a color. The polygon is oriented, which means that from it one can construct its **normal function** $Ax + By + Cz + D$ which is 0 on the face and with the unit vector $[A, B, C]$ pointing outwards. The normal function is used to test visibility and also to determine shading. If $[r, g, b]$ is the face's color, then the displayed color is calculated in the following way: let d be the dot-product of the light source and the normal vector $[A, B, C]$. Because both are normalized, this lies in the range $[-1, 1]$—1 if the light source is perpendicular to the face and -1 if it is opposite. Thus $(1 + d)/2$ lies in $[0, 1]$, where 0 means no light. Finally,

what I call a fudge function (in the form of a Bernstein polynomial) is applied to this to get a number s again in $[0, 1]$, and the color displayed is $[sr, sg, sb]$. Crude, but adequate. Here are the relevant functions:

- `convexsurface(f)`

f is an array of faces

- `paint(s)`

s is a surface.

- `face(p, c)`
  `face(p, c, ...)`

Here p is an array of 3D polygons, $c = [r, g, b]$. The `...` can be anything. For example, to get smooth shading, add an array of normals to the vertices to get a smooth convex surface.

- `reverse(f)`

Changes the orientation of the face by multiplying its normal function by -1.

- `setshading(f, y)`

Here f is a face, y is an array of at least 3 numbers in $[0, 1]$, parametrizing a Bernstein polynomial.

Here is a sample, the rotating square again.

```
from piscript.PiModule import *
import math

init3d("face", 200, 100)

seteye([0,0,10,1])
setlight([-1,1,0.5,0])
f0 = face([
    [0,0,0],
    [1,0,0],
    [1,1,0],
    [0,1,0],
],
[1, 0, 0])
f1 = face([
    [0,0,0],
    [0,1,0],
    [1,1,0],
    [1,0,0],
],
[0, 0, 1])

c = convexsurface([
    f0, f1
])

for i in range(36):
    beginpage()
    center()
    scale(64)
    translate(0,-0.5)
```

```
        paint(c)
        endpage()
        rotate3d([0,1,0], math.pi/18)

    finish()
```

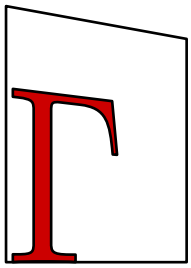
Note the reversal of orientation in the second face. In 3D, it is often important to choose the exact position of a figure in order to see it clearly. The kind of primitive animation done here helps one decide which is best. After a choice has been made, one can reduce the number of loops to 1.

- `mappedface(p0,p1,p2)`

Here the points p are affinely independent points in 3D. This returns a structure with (a) an affine map taking $(0,0)$ to $p0$, $(1,0)$ to $p1$, $(0,1)$ to $p2$, (b) the normal function corresponding to the triangle $p0, p1, p2$ and a corresponding visibility test `isvisible(eye)`; (c) a function `makepath(p)` which takes a 2D Path as argument, and maps it to a 3D path acording to the map. In other words, a mapped face is a way to associate paths in 3D to those in the plane—it *maps* such paths into 3D. The following code will, on my system, draw the character with index 0 from the font `cmr10`, which happens to be Γ, in 3D.

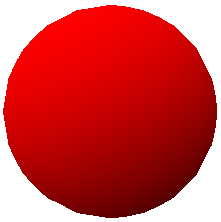
```
from piscript.PiModule import *

f = font("/usr/share/texmf/fonts/type1/bluesky/cm/cmr10.pfb")
c = charpath(f, 0)

init3d("C3d", 100, 100)
seteye([0,0,40,1])
A = 0
setdeg()
for i in range(72):
    gsave3d()
    rotate3d([0,1,0], A)
    beginpage()
    center()
    translate(-48,-48)
    scale(9.6)
    gsave3d()
    scale3d(1.0/100, 1.0/100, 1.0/100)
    mf = mappedface([0,0,0],[1,0,0],[0,1,0])
    e = geteye()
    newpath()
    mf.makepath(c)
    if mf.isvisible(e):
        fill(0.8,0,0)
        stroke(0)
    else:
        fill(1,1,0)
        stroke(0)
    grestore3d()
    scale3d(10,10,10)
    newpath()
    mf.makepath([
        Moveto, 0,0,
        Lineto, 1,0,
        Lineto, 1,1,
        Lineto, 0,1,
```

```
            Closepath
        ])
        stroke(0)
        endpage()
        A += 5
        grestore3d()

finish()
```



• `smoothconvexsurface(f)`

A smooth surface is again an assembly of polyhedral surfaces, but now each vertex is assigned a normal and a color. These are used to interpolate, using the `shfill` of PostScript, to color each face. The only example of this currently done is the sphere. `f` here is an array of triangles `[p, c, n]` where `n` is the array of the three unit normals at the vertices. The only example I have of this is a unit sphere:



• `sphere(c, n)`

Here `c` is the color, and $1 \leq n \leq 4$ is an integer that controls how many times the icosahedron is subdivided to make the spherical polyhedron. Higher values for n mean a smoother surface.

The 3D package in $\pi\mathcal{S}$ is so far pretty simple. Dealing with convex bodies is pretty easy for this style of work, but without a z-buffer effects such as transparency are hard, even nearly impossible.

Part 4. Miscellaneous

Stuff that didn't seem to fit elsewhere.

17. Vectors

The module `piscript.PiModule` contains a Python class `Vector`, which is also available as separately as the module `piscript.Vector`. It defines a Python class `Vector`, and also a number of simple geometric operations and functions that are very useful for drawing. It is algebraic notation that makes Vectors convenient.

Here is some sample usage:

```
from piscript.PiModule import *

u = Vector(0, 1)
v = Vector([2,-1])
w = u + v
print w
```

The operations available on Vectors are:

`u + v`	vector addition
`u - v`	vector subtraction
`u[i]`	the i-th coordinate
`-v`	the negative of v
`u*c`	scalar multiplication by c
`u*v`	dot product with v
`u/c`	scalar division by c
`u[i]`	the i-th coordinate
`u.cross(v)`	the cross product (of 3D Vectors)
`u.length()`	the Euclidean length
`abs(u)`	also the Euclidean length
`len(u)`	the length as an array—its dimension
`u.arg()`	the angle of the direction of the 2D vector u, in radians

In these, the first operand u must be a Vector, but the second operand v can be just an array. When `str` or `print` is called on a Vector, you will see the proper format, such as `[ 1, 2 ]`. As mentioned in the command descriptions, it is allowable to use Vectors as arguments in almost all methods whenever coordinate arrays are now acceptable.

There are also a number of static methods in the Vector class, which I discuss in an appendix. There is a related Module `VectorUtils` that contains the same procedures. This is mostly to deal with older πS programs.

18. Addressing PostScript directly

There are a number of operations that interact more directly with the PostScript file.

- `importPS(f)`
- `importEPS(f)`

Loads the PostScript file f into the output file. The difference between the two is that the second encapsulates the loaded file from its environment. This is how you would include a photograph, for example—load the .jpg file into `gimp`, save it as .eps, and then import it. The other does no encapsulation, and is largely intended for loading fonts, where you want the imported file to affect the environment.

If your PostScript viewer seems to choke on imported `.eps` files, particularly ones made from photographs, try turning on the option that pays no attention to end of file markers. In `gv`, for example, this is the `gv` option "Ignore EOF". It is accessed by opening the menu `State/gv Options ...`

You can import images, for example photographs, in your figures, but you will have to convert them to PostScript images first, say with the commonly available program GIMP.

```
from piscript.PiModule import *

init("picture", 198, 202)
beginpage()

gsave()
translate(-2.5,-3)
scale(0.18)
importEPS("koala.eps")
grestore()

setfont("/Helvetica-Bold", 8)
moveto(3.5,90)
setcolor(0.8,0,0)
show("KOALA")

endpage()
flush()
```

produces

- `eol()`

Adds a blank line to the output file. Useful for making that file readable by humans, in case—just in the very unlikely, gosh! almost impossible, event—that something really goes wrong.

- `putPS(s)`

Adds an arbitrary string to the output file. This is a last resort, allowing you to do fine work in PostScript if necessary. The good news is that this feature really doesn't exclude any thing. The bad news is that you have to know PostScript fairly well in order to use this feature. So this operation is here only for the cognoscenti (or rather, since I am in Berlin as I write this, for the Feinschmeckern).

- `comment(s)`

Adds a string as PostScript comment. Again, makes the output more readable for humans, which is sometimes helpful in seeing what goes wrong. But reading PostScript files also means you have to know how to program in PostScript.

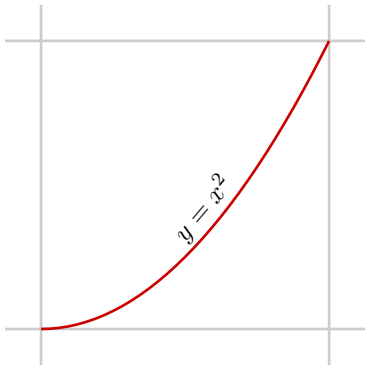
19. Differences from PostScript

Many of the commands have names almost the same as the ones they call in PostScript, but sometimes the behaviour is different. One major reason for doing this is to make it easy to maintain `gsave/grestore` pairings. Sometimes it is just convenience. For example, the `fill` command has an option in which the fill color is specified in the command, although this is not the default. The advantage of this is that here the color change is undone automatically after the fill whereas in PostScript it is not. Also, it is very common in mathematical work to both fill the inside of a region and then afterwards draw its boundary. In PostScript this is done awkwardly with a `gsave/grestore` pair, since in most PostScript graphics it is less common to both fill and stroke a path. The necessary `gsave/grestore` commands are built into the πS `fill` and `stroke` commands, thus getting a mild gain in convenience at a small cost in efficiency.

Other differences: (a) Angles in PostScript are in degrees, in πS either radians or degrees. (b) In PostScript the commands `fill` and `stroke` have an indeterminate effect on the current path in PostScript, maybe destroying it or maybe not. In πS they do not destroy it. This makes it especially important to start paths with `newpath()`. (c) In PostScript line widths are scaled when the coordinate system is. But in πS the line width is maintained at constant $1/72''$ unless it is scaled with `scalelinewidth` or `setlinewidth`. In other words, πS thinks of a line as a mathematical line, having thickness only to make it visible.

20. An example

To produce



write

```
from piscript.PiModule import *
init("squaregraph", 144, 144)

settexenv("latex")
setdeg()
beginpage()

center()
scale(108)
translate(-0.5, -0.5)
margin = 0.125
newpath()
moveto(-margin, 0)
lineto(1+margin, 0)
moveto(1, -margin)
```

```
lineto(1, 1+margin)
moveto(1+margin, 1)
lineto(-margin, 1)
moveto(0, 1+margin)
lineto(0, -margin)
stroke(0.8)

# now draw the graph
setcolor(0.8, 0, 0)
N = 32
t = 0
dt = 1.0/N
newpath()
moveto(t, t*t)
for i in range(N):
    t += dt
    lineto(t, t*t)
stroke()

gsave()
setgray(0)
gs = revert()
translate(gs.transform(0.5, 0.3))
rotate(50)
place(texinsert("$y = x^2$"))
grestore()

endpage()
finish()
```

I could have used `graph` for this, but I thought you should see what that procedure does underneath the hood. Thus you can see easily how to draw a planar curve parametrized by a function $f(t) = \big(x(t), y(t)\big)$. Other examples can be found in the directory `piscript/examples`. These can also be found at

```
http://www.math.ubc.ca/~cass/piscript/examples
```

21. Fooling around with the bounding box

The bounding box is normally set in the initialization, but it is possible to change it dynamically.

- `setbbox(bbox)`

Here `bbox` is an array of four numbers, which represent the corners of the desired bounds specified *in current coordinates*. These will be converted to integers, expanding outwards. This command can be inserted anywhere in a program, and will redefine the bounding box. It will often be used in conjunction with the argument "noclip" for `beginpage`, because otherwise the image is clipped according to the bounding box current at the time `beginpage` is called.

- `currentbbox()`

Returns the current bounding box in terms of current coordinates.

- `boundary()`

Equivalent to `bbox(currentbbox())`—i. e. it constructs the current figure's boundary as a path. This is here largely just to handle legacy code.

- `width()`
- `height()`

Returns the nominal width and height of the bounding box, $urx - llx$ and $ury - lly$ (in current units). For these to be useful, the coordinate system should be rectangular.

- `envelope(a, b)`

Returns the smallest bounding box containing the two boxes a and b, in terms of current coordinates.

- `cliptobbox()`

Clips to the interior of the current bounding box.

The commands that set the bounding box have to be used carefully, especially in combination with `beginpage`. The principal purpose of having them is to set the bounding box in terms of familiar units, or as one draws and sees that a larger or smaller bounding box would be appropriate. However, normally beginning a page will automatically clip to the current bounding box, and if one changes the bounding box this clipping path will stay in effect. This can lead to strange and probably unwanted effects. Normally, one should call `cliptobbox()` immediately after setting the bounding box. Above all, you probably do not want to draw outside the eventual bounding box, although there are times when this is desirable.

Using the standard `dvips` program in a TEX file, you can force clipping from within the TEX file, by calling `\epsfclipon`, and restore the default (no clipping) by calling `\epsfclipoff`. See

`http://www.pd.infn.it/TeX/doc/html/dvips/dvips_5.html`

22. Coming Real Soon Now to a screen near you

One apparently controversial question is "How to do commutative diagrams" Are they part of the text or graphics images? The answer is, neither. Here, it is especially important that the text in the diagrams harmonize with the enclosing text, but it is also true that good commutative diagrams require the flexibility in design that one has only in graphics. Another, more minor, problem concerns what kind of arrows you want in these diagrams. The harmonization business, at least, can be dealt with by suitable TEX prefix. As for arrows—well, you're free to roll your own! Or use `Xy-pic`.

Currently, I do not handle any PostScript fonts except those fonts of Type 1 in TEX. This applies in particular to the `.gsf` fonts that come with GhostScript. This also shouldn't be hard to do, but the methods for dealing with other fonts are not the same as they are for Type 1 fonts, so there is something non-trivial to be done.

There are many hidden utilities in `Type1Font` for manipulating fonts, and eventually I should make it possible to have $\pi\mathcal{S}$ load `.pfb` fonts and convert them to `.pfa` format for direct use with the `show` command, and in fancy ways.

Is it worthwhile to improve the 3D package? The most important thing missing are techniques to deal with collections of several objects, which have to be drawn from back to front, and in some cases prepared for that by chopping objects up. There are many great 3D tools around, but most of them are far heavier than most mathematical exposition calls for.

23. General comments

The disadvantages of πS include above all verbose input and output. The verbose input is something one gets quickly used to, and it goes hand-in-hand with cut-and-paste programming, since one can often just copy large chunks of code from one part of a program to another. The verbose output is mildly annoying. The major cause of this is that loops are unrolled in the output, instead of being part of loops in PostScript itself. A solution to this problem is to put the interface between PostScript and πS somewhere nearer PostScript, but I'll not do that soon.

It may seem that there are a huge number of commands in πS. In my experience it doesn't take long to produce simple diagrams quickly, and as for the rest, they grow on you. One thing to keep in mind is that you can define your own commands to make repetitive task easier. You can even modify the basic code of πS, but I ask that you not do that—these things have a tendency to spread like Asian flu, and could cause a lot of confusion sooner or later. So if you do want to modify my code, rename it!

The advantages of πS are these: (a) the PostScript output runs fast and is quickly rendered into PDF (this last is probably true because programs that convert PostScript to `.pdf` are also largely concerned with unrolling loops, and in the output pf πS there are, as I have just pointed out, no loops); (b) the user has virtually complete control over the graphics (this is not unrelated to verbosity, is it?); (c) TEX inserts are absolutely painless; (d) programming in Python is pure pleasure (and in particular, for PostScript programmers like me, the error handling is marvelous). I have worked hard to make actual errors in PostScript very unlikely. Not impossible. But if they do occur you will get a shock from your PostScript viewer—error messages will be terrifying. Refer to Chapter 1 of **Illustrations** for some advice on how to keep your sanity.

I want to repeat something I mentioned briefly earlier. When working in an operating system that supports the `make` utility, using πS to generate `.eps` files can be remarkably convenient. You can configure `make` to understand the dependency of `.eps` files on `.py` files. This requires that the output `.eps` file have the same prefix as the `.py` files, which has be taken into account in the initialization in the `.py` file. Then, any changes in `.py` file will be automatically transferred to the corresponding `.eps` files if you type `make` in the appropriate directory.

Another problem with πS is one frequently raised, the compatibility of fonts in figures with that in enclosing text. This compatibility is one of the principal of the packages that do figures from within TEX. I myself see no advantage to this compatibility, but emotions can run high over this matter.

Part 5. Coordinate systems

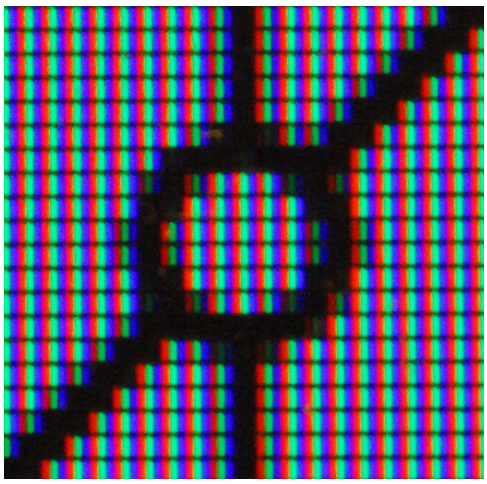
In order to use πS efficiently it is important to understand coordinate systems. This is standard fare in mathematics courses, but what is interesting about the way they are dealt with in both PostScript and πS is that internally they work with an extra dimension. Understanding how and why this is done is not required at any point in using πS, but it might be useful if you want to develop your own graphics programs.

24. Coordinates in 2D

Making a figure with πS involves writing down a lot of instructions involving coordinates. But in order to see what your program actually produces, these instructions and numbers must be transformed at some point into instructions to a piece of hardware, for example either your computer display or through your printer onto paper. Some kind of translation process is involved. Here is a πS program that draws two lines through the origin at the centre of the window, and then puts a small circle at the origin:

```
center()
scalelinewidth(0.75)

newpath()
moveto(-100,-100)
lineto(100, 100)
moveto(0,-100)
lineto(0, 100)
stroke()

newpath()
circle(2)
fill(1)
stroke(0,0,0)
```

and here is what a very close look at the center of my display window looks like:



Many interesting things are visible here. The most interesting is perhaps the effect of what is called *anti-aliasing*, by which sharp breaks in shade are rendered as gradients in order to give the illusion of smoothness to an otherwise 'jaggy' line. The point at the moment is that there is a translation involved, from the coordinate system in which I am programming to the coordinate system the hardware uses. Now almost all graphics hardware these days renders graphics in terms of pixels—very small regions of your screen that represent essentially one

minimal display unit. The basic unit of length on such a display is naturally the width of one pixel. In addition to choosing this dimension, other choices must be made—what directions for x and y axes, and the location of the origin. On my display window, which simulates an $8\,1/2" \times 11"$ page, for example, the origin is at the upper left of the window, x increases to the right, and y increases as you go down the screen. In addition, there are about 75 pixels to one nominal linear inch, which means that each point is about $75/72 \sim 1.04$ pixels. Recalling that the origin in my program is at lower left when I start, one deduces that the point which is (x, y) in my program maps to the point which is $(x_{\text{pixel}}, y_{\text{pixel}})$ in the window, where

$$\begin{aligned} x_{\text{pixel}} &= \quad 1.041667x \\ y_{\text{pixel}} &= -1.041667y + 825 \end{aligned}$$

For example, $(0, 0)$ maps to $(0, 825)$ and since $825 = 11 \cdot 75$, this is indeed at lower left.

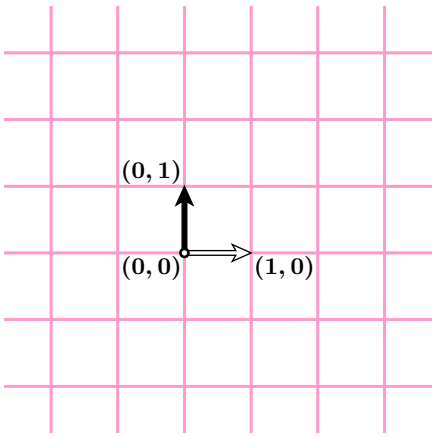
25. Affine transformations

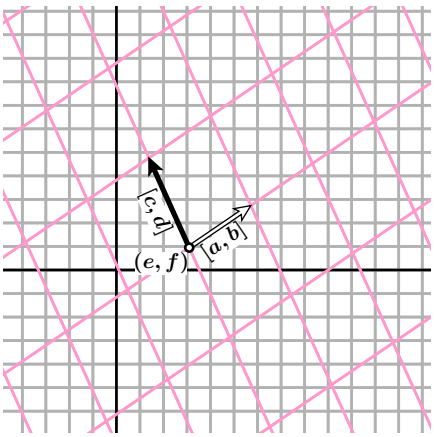
The coordinate systems used in most graphics programs are **affine**. Each coordinates system is determined by a coordinate frame, which amounts to a choice of origin as well as two vectors equal to the unit displacements along the x, y axes. The relationship between two affine coordinate systems is specified by an **affine transformation**. Geometrically, this means that a straight line in one is transformed to a straight line in the other. Algebraically, it means that the coordinates (x, y) and (x_*, y_*) are related by equations

$$\begin{aligned} x_* &= ax + cy + e \\ y_* &= bx + dy + f\,. \end{aligned}$$

An affine transformation is stored internally in PostScript and $\pi\mathcal{S}$ as an array $[a, b, c, d, e, f]$. (I do not know the derivation of the term 'affine'. My dictionary tells me that 'affine' is an English word coming from the Latin word 'affinis', which means 'related'. So much for that.)

The individual coefficients in this expression can be geometrically interpreted. (a) If $(x, y) = (0, 0)$ then $(x_*, y_*) = (e, f)$. So (e, f) is the point the origin gets mapped to. (b) The point $(1, 0)$ is transformed to $(a+e, b+f)$, so (a, b) is what the vector from $(0, 0)$ to $(1, 0)$ is transformed to—each shift in (x, y) by $[1, 0]$ gives rise to a shift in (x_*, y_*) by $[a, b]$. (c) Similarly, $[c, d]$ is what the relative displacement $[0, 1]$ corresponds to. (I am making a distinction between *points* (x, y), which are characterized by *location*, and *vectors* $[dx, dy]$, which are characterized by relative displacement—have direction and magnitude but not location, as they told you in physics class. Given a coordinate system, the two are confounded because location can be seen as a displacement from the origin.)





Here, we'll be concerned with different coordinate systems on the same plane rather than on different ones. That's because using coordinate changes effectively is a crucial technique in making $\pi\mathcal{S}$ useful.

Each coordinate system is completely determined by a single unit frame F embedded into the plane, which is a parallelogram with one corner and two adjacent edges labelled. If we are given a coordinate system, a frame F is equivalent to a point specified by its coordinates, together with two vectors $[a, b]$ and $[c, d]$ determining the edges. But now choose the new coordinate system whose unit frame is F. The relationship between a point's old coordinates and its new ones is

$$x_{\text{old}} = ax_{\text{new}} + cy_{\text{new}} + e$$
$$y_{\text{old}} = bx_{\text{new}} + dy_{\text{new}} + f\,.$$

For example, the new origin is (e, f) in the old coordinate system.

26. Changing coordinates

When πS starts up, the coordinate system is the default. The origin is at lower left of the page, and the coordinate frame is a square, $1/72''$ on a side. As a πS program proceeds, the coordinate system changes, and πS itself keeps track of the matrix transforming one set of coordinates to another. *When we change coordinates, how does the current transformation matrix change?* The most elegant answer to this question involves an interesting move from 2D into 3D.

The equation relating coordinates can be put into matrix form:

$$\begin{bmatrix} x_{\text{old}} \\ y_{\text{old}} \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}\,.$$

The shift by (e, f) gets treated differently from the other parts of the transformation. But we can rewrite this equation in an interesting way—suppose we embed the 2D plane in 3D by mapping (x, y) to $(x, y, 1)$. In other words, we shift the plane $z = 0$ to $z = 1$. Then the above equation can be rewritten

$$\begin{bmatrix} x_{\text{old}} \\ y_{\text{old}} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \\ 1 \end{bmatrix} = A \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \\ 1 \end{bmatrix}\,.$$

The coordinate transformation is implemented now by a familiar matrix multiplication. This trick makes the combination of several coordinate changes easy to keep track of. If we also have

$$\begin{bmatrix} x_{\text{default}} \\ y_{\text{default}} \\ 1 \end{bmatrix} = T \begin{bmatrix} x_{\text{old}} \\ y_{\text{old}} \\ 1 \end{bmatrix}$$

then

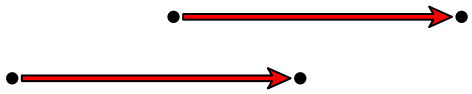
$$\begin{bmatrix} x_{\text{default}} \\ y_{\text{default}} \\ 1 \end{bmatrix} = T \begin{bmatrix} x_{\text{old}} \\ y_{\text{old}} \\ 1 \end{bmatrix} = TA \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \\ 1 \end{bmatrix}\,.$$

So the new current transformation matrix is the matrix product TA.

27. Points and vectors (again)

The translation from 2D to 3D helps make the difference between points and vectors more visible.

I recall: a **point** is a position, a **vector** is a relative position. In other words, a vector measures the *difference in position* of two points. A vector has no intrinsic location—in the following figure, the two arrows represent the same vector.



In writing, I try to distinguish points (x, y) from vectors $[x, y]$, but this is not always feasible since programming languages see both as just an array of two numbers. This confusion has a meaning—the numbers have meaning only if a coordinate system has been chosen, and this means in particular an origin. Given an origin, every point is associated to the vector representing displacement from that origin. And conversely, every vector is associated to the point you get by displacing the origin. But unless the choice of origin has real significance, this correspondence has no real significance—if we change the origin, points get attached to different vectors.

The operations strictly allowed on points and vectors are quite different, although of course when they are both treated as arrays the distinction fails in practice. On points there are very few operations: basically, all we can do is shift a point by a vector, and we can take the difference of two points to get a vector. But it does not make sense to add two points. As for vectors, we can add them or scale them.

We can take the dot product of two vectors, but this isn't intrinsic—it depends on the units of length we have chosen, and how we calculate it depends on the coordinate system at hand.

There are other objects of geometrical significance—lines and functions. They also are assigned coordinates if a coordinate system is chosen. The functions we'll consider here are linear

$$f(x, y) = ax + by$$

and affine

$$f(x, y) = ax + by + c\,.$$

The way to distinguish these is that *a linear function may be intrinsically applied to vectors, and an affine one to points*. There is a relaton between the two—to an affine function $ax + by + c$ is associated its oldgradient function $\nabla f(x, y) = ax + by$ which satisfies

$$f(P + V) = f(P) + \nabla f(V)$$

for any point P, vector V. Given a coordinate system, points and vectors are confused, since linear functions may be identified with affine functions vanishing at the origin.

One curious feature is that if we shift 2D points into 3D, the evaluation of an affine function at points becomes the evaluation of a linear function:

$$a \cdot x + b \cdot y + c = [a, b, c] \bullet [x, y, 1]\,.$$

This is useful in understanding how coordinate changes affect functions.

How does a linear coordinate change affect a linear function? Intrinsically, the function assigns numbers to points, but the formula for that function depends on a choice of coordinate system. For example, consider the function $x + y$. Change coordinates' $x_{\text{new}} = 2x_{\text{old}}$, $y_{\text{new}} = 3y_{\text{old}}$. The point that is $(1, 1)$ in the original system becomes $(2, 3)$ in the new one. The function doesn't change its value at the point, however, so its expression in the new system is $x_{\text{new}}/2 + y_{\text{new}}/3$.

A linear function may be expressed as a matrix product, if we express the coefficients of the function as a row vector:

$$ax + by = \begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} .$$

Suppose we make a linear coordinate change

$$\begin{bmatrix} x_{\text{old}} \\ y_{\text{old}} \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} = A \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} .$$

Let f be the function expressed in terms of coordinates $(a_{\text{old}}, b_{\text{old}})$ in the original system. We have

$$a_{\text{old}}x_{\text{old}} + b_{\text{old}}y_{\text{old}} = \begin{bmatrix} a_{\text{old}} & b_{\text{old}} \end{bmatrix} \begin{bmatrix} x_{\text{old}} \\ y_{\text{old}} \end{bmatrix} = \begin{bmatrix} a_{\text{old}} & b_{\text{old}} \end{bmatrix} A \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix} = \begin{bmatrix} a_{\text{new}} & b_{\text{new}} \end{bmatrix} \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \end{bmatrix}$$

if we set

$$\begin{bmatrix} a_{\text{new}} & b_{\text{new}} \end{bmatrix} = \begin{bmatrix} a_{\text{old}} & b_{\text{old}} \end{bmatrix} A .$$

The same thing happens with affine functions:

$$\begin{bmatrix} a_{\text{new}} & b_{\text{new}} & c_{\text{new}} \end{bmatrix} = \begin{bmatrix} a_{\text{old}} & b_{\text{old}} & c_{\text{old}} \end{bmatrix} A$$

if A is the 3×3 matrix representing an affine change of coordinates.

If we identify 2D points with 3D ones, an interesting happens: the difference us

$$(x_1, y_1, 1) - (x_2, y_2, 1) = (x_1 - x_2, y_1 - y_2, 0) .$$

This suggests identifying 2D vectors with 3D points whose last coordinate is 0.

28. Homogeneous coordinates

An affine function $ax + by + c$ determines the line $ax + by + c = 0$. What are the coordinates of the line? If we multiply the equation through by a non-zero scalar, say 2 or -1, we get the same line. So we take the coordinates of the line to be (a, b, c) but with the proviso that (ya, yb, yc) is in some sense the same set of coordinates. These are called **homogeneous coordinates**—determined only up to non-scalar multiplication. This idea can be refined slightly—restrict the scalar multiplication to positive scalars. Thus (a, b) is not the same as $(-a, -b)$. This also has some intrinsic meaning, because it allows us to distinguish sides of the line: $ax + by > 0$ and $ax + by < 0$. We might call a line together with a choice of sides an **oriented line**.

It also makes some sense to consider homogeneous coordinates for vectors, too: then the vector $[a, b]$ corresponds to the 3D vector $[a, b, 0]$, and if we allow multiplication by a positive scalar we still preserve the **direction** of the vector.

What about for points? It turns out that even with points using homogeneous coordinates makes sense.

In 3D we map the point (x, y, z) to $(x, y, z, 1)$ or now, in effect, to all the points (cx, cy, cz, c) for $c > 0$. I am going to give just one application of this construction to a problem that arises frequently in 3D graphics.

> *Fix a point P and a plane $f(x, y, z) = Ax + By + Cz + D = 0$. For each point $Q \neq P$, there exists a unique line through P and Q, and if it is not parallel to the plane it will intersect the plane at a unique point R. What is R?*

In effect, the point R is the projection of Q onto the plane from the point P.

The problem is basically simple. The line from P through Q can be parametrized $P + t(Q - P)$, so we want to find t such that

$$f\big(P + t(Q - P)\big) = 0$$

If $\nabla f = [A, B, C]$ so that $f(P) = \langle \nabla f, P\rangle + D$, this leads to

$$f(P) + t\langle \nabla f, (Q - P)\rangle = 0$$
$$t = -\frac{f(P)}{\langle \nabla f, (Q - P)\rangle}$$
$$R = P - \frac{f(P)}{\langle \nabla f, (Q - P)\rangle}\,(Q - P)$$

This last equation still holds if we push P, Q, R into 4D with last coordinate 1*, since the last coordinate of $Q - P$ is then* 0. Even better, we interpret the coefficients for the plane as a 4D point (A, B, C, D). This seems rather natural, since

$$f(P) = Ax + By + Cz + D = (A, B, C, D)\bullet(x, y, z, 1)\,.$$

The equation for R becomes now

$$R = P - \frac{f(P)}{\langle f, (Q - P)\rangle}\,(Q - P)\,.$$

and if we take into account that multiplying homogeneous coordinates by non-zero scalar is allowable, we get

$$R \sim \langle f, (Q - P), \rangle P - f(P)(Q - P) = f(Q)P - f(P)Q\,,$$

which is surprisingly elegant. Of course to bring R back to 3D it is necessary to divide through by the 4-th coordinate to make it 1. This coordinate is $\langle f, (Q - P)\rangle$. It will be 0 precisely when $f(P) = f(Q)$, which means that $Q - P$ is parallel to the plane $f = 0$ and the projection is undefined anyway.

There is a second version of this problem which is also handled nicely by the formula. Suppose we take P to be very far away. This means that $r = \|P\| = \sqrt{x^2 + y^2 + z^2}$ is very large. Using homogeneous coordinates, $P = (x, y, z, 1)$, and this is equivalent to $(x/r, y/r, z/r, 1/r)$. As $r \to \infty$, this has a limit with last coordinate 0. This suggests the idea that more generally *points at infinity in 3D are equivalent to 4D points with last coordinate* 0, which will be useful to keep in mind. The original problem still makes sense for such infinite points—in that case we are looking at projection onto the plane from a certain direction indicated by the coordinates of P. The really nice thing is that *the formula above for projection remains valid for points at* ∞.

In order to really appreciate this formula, and to see what interesting use it can be put to, it is necessary to understand how PiScript implements 3D coordinate changes. First of all it maintains a 3D graphics stack on which the 3D graphics state is held. This includes mostly a matrix T that tells how to transform the current user coordinates—the ones that go as arguments to commands such as `moveto3d(x,y,z)`—to the default 3D coordinates it starts with. It also stores some information about how then to transform points in default coordinates into a point in 2D to be plotted and seen.

Internally, PiScript works with homogeneous coordinates. The matrix T is therefore 4×4, and the matrix that specifies how to go from 3D to 2D is 4×3. The way the transform works is that $v = (x, y, z, 1)$ is mapped to Tv in default coordinates. Evey coordinate change corresponds to a 4×4 matrix A, and its effect is to change T to TA. For example, a translation by (a, b, c) means

$$(x_{\text{old}}, y_{\text{old}}, z_{\text{old}}) = (x_{\text{new}} + a, y_{\text{new}} + b, z_{\text{new}} + c)\,,$$

which can be written

$$\begin{bmatrix} x_{\text{old}} \\ y_{\text{old}} \\ z_{\text{old}} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \\ z_{\text{new}} \\ 1 \end{bmatrix}\,,$$

so for a translation by (a, b, c)

$$A = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}\,.$$

Rotations and scale changes are similar.

Now one surprising thing. Suppose we want to draw shadows of an object in 3D, shadows onto some plane from a light source at the point P. This can be effected also by a kind of coordinate change! As we carry out commands like `moveto3d(Q)` we really want to move to the projection of Q onto the plane of shadows. But the formula taking Q to $f(Q)P - f(Q)P$ depends linearly on Q, and can be expressed by a matrix. If $P = (x_P, y_P, z_P, 1)$ and $Q = (x_Q, y_Q, z_Q, 1)$ then the projection is

$$R = f(Q)P - f(P)Q = (Ax_Q + By_Q + Cz_Q + D)(x_P, y_P, z_P, 1) - f(P)(x_Q, y_Q, z_Q, 1)\,.$$

The matrix associated to this has as its columns the transforms of the basis vectors $Q = (1, 0, 0, 0)$ etc. and it is here

$$\begin{bmatrix} Ax_P - f(P) & Bx_P & Cx_P & Dx_P \\ Ay_P & By_P - f(P) & Cy_P & Dy_P \\ Az_P & Bz_P & Cz_P - f(P) & Dz_P \\ A & B & C & D - f(P) \end{bmatrix}.$$

Thus we are led to introduce a command

- `project(f, P)`

Here $f = [A, B, C, D]$ and $P = (x, y, z, w)$ with $w = 1$ for points and 0 for parallel projections. This effects shadow drawing by a coordinate change. It differs from other coordinate commands in one important way—when coordinate changes are made, the inverse of the matrix T is also calculated—$T^{-1} \mapsto A^{-1}T^{-1}$. But projection is not invertible, and this calculation cannot be done. What this means in practice is that you cannot make more coordinate changes on top of a projection. Be sure to encapsulate `project` by `gsave/grestore`.

Part 6. Mathematics matters

29. Shading

I'll first look at how to darken or lighten the color of a surface in 3D, according to how it is affected by a light source. As I have said before, the purpose of this in πS is not to make scenes look realistic, but just to help the eye understand what it is seeing.

The light source is specified by a 4D vector $L = [x, y, z, 0]$, where the 0 signifies that it is at ∞. This vector is normalized so as to have length 1. The orientation of $[a, b, c]$ matters, so this is not strictly speaking a homogeneous vector; equivalence is ruled by positivce scalar multiplication. To a polygonal fragment of a surface is associated its normal function $\nu = [a, b, c, d]$, characterized by the property that the affine function $ax + by + cz + d$ vanishes on the surface, and the fragment is oriented so that $[a, b, c]$ points outwards. To determined darkening, we start by computing the dot product $d = L \bullet \nu$. The cosine rule tells us that if θ is the angle between L and ν then

$$\cos\theta = \frac{L \bullet \nu}{\|L\| \, \|\nu\|} = d\,,$$

so that $-1 \leq d \leq 1$. It is -1 when L and ν are opposite, in which case the surface should be dark, and it is $+1$ when the two are the same, in which case it should be bright. In general, we should expect the brightness to be a monotonic function of d, and in the range $[0, 1]$ so as to give color parameters in the right range.

So we must ask, how can one offer a good choice of monotonic functions

$$f\colon [-1, 1] \to [0, 1]\,?$$

The first and simplest step is to change d to $(1 + d)/2$, and then ask for monotonic functions from the unit interval $[0, 1]$ to itself.

I have chosen the class of **Bernstein polynomials** for this purpose. A Bernstein polynomial of degree n is one of the form

$$B_y(x) = \sum_0^n y_i \binom{n}{i} (1 - x)^i x^i\,,$$

where $y = (y_i)$ is an array of arbitrary coefficients. These were invented by the Russian mathematician Sergei Bernstein in the early 20th century to provide a construction proof of a sequence of polynomials approximating an arbitrary continuous function on the unit interval (a theorem originally due to Karl Weierstrass in a more abstract form).

Theorem. *If $f(x)$ is an arbitrary continuous function on $[0, 1]$, then the functions*

$$f_n(x) = B_y(x) \quad (y = [f(0), f(1/n), f(2/n), \ldots, f(1)])$$

converge to f as $n \to \infty$.

In other words, arbitrary continuous functions amy be approximated by Bernstein polynomials, so that using them has a chance of not being a practical restriction.

These polynomials have for our purposes a number of useful properties:

- $B_y(0) = y_0$;
- $B_y(1) = y_n$;
- *If y has length $n+1$ then* $B'_y(x) = nB_{\Delta y}(x)$.

where Δy is the array of differences $y_{i+1} - y_i$. As a consequence:

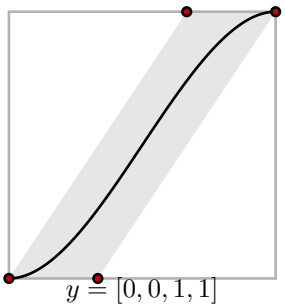
- $B'_y(0) = n(y_1 - y_0)$;
- $B'_y(1) = n(y_n - y_{n-1})$.

Thus the graph of $B_y(x)$ starts at $(0, y_0)$ and heads towards $(1/n, y_1)$; ends at $(1, y_n)$ and comes from $(1 - 1/n, y_{n-1})$. Since

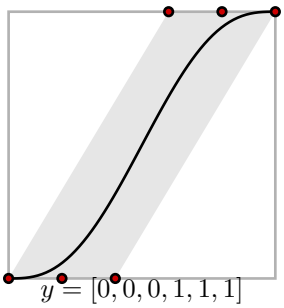
$$\big((1-x)+x\big)^n = \sum (1-x)^i x^{n-i} \binom{n}{i} = 1$$

the value of $B_y(x)$ is a weighted sum of the coefficients y_i, and the graph of B_y between 0 and 1 is contained in the convex hull of the points $(i/n, y_i)$. Putting all these things together, we see that if the coefficients y_i are a non-decreasing sequence and lie in $[0, 1]$ then $B_y(x)$ will start at $0 \le y_0$ and increase to $y_n \le 1$. Furthermore, Bernstein's theorem tells us that we are not sacrificing any practical generality by restricting ourselves to Bernstein polynomials.

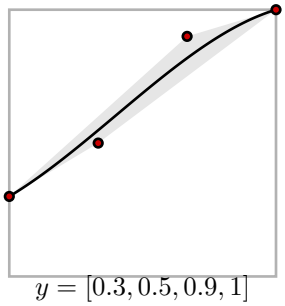
The figures below display the graphs of some Bernstein polynomials, with the y_i and the convex hull also indicated.



$y = [0, 0, 1, 1]$



$y = [0, 0, 0, 1, 1, 1]$



$y = [0.3, 0.5, 0.9, 1]$

The default shading for convex surfaces in $\pi\mathcal{S}$ is $y = [0.3, 0.5, 0.9, 1.0]$. It can be changed with

- `setshading(s,y)`

where s is a convex surface, y a shading array. The surface you see will be matte—no shiny billiards here. I remind you that there is also a version `setshading(f,y)` with f a face.

In addition to what I have discussed here, there is a module `Bernstein` distributed with $\pi\mathcal{S}$ that contains a small number of useful functions related to Bernstein polynomials.

Part 7. Advice on illustrating mathematics

With πS you have at hand a tool for producing good mathematical pictures of a very high quality, one that is as simple to use as it could be. But as when you have to get used to driving a new car, what you have might not be apparent at first. Producing good figures is not entirely straightforward. It requires imagination and care.

- *Do not overload diagrams.*

- *Reduce visual clutter and—what is not quite the same thing—eliminate distraction.* Put in only what the diagram really needs to make its point. Tone down components that just add context. One eccentric idea is that, in my view, vertex labels are always clutter. In the year 300 B.C. they might have been a useful and novel innovation, but in the modern world there are more useful tricks to try.

- *Highlight components that are central to the current discussion.* If necessary, repeat a diagram several times, but with different components emphasized. This is a variant of what Tufte calls small multiples.

- *The figures themselves should tell a story.* Coordination between text and illustration is surprisingly tricky, and ideally the two should be as independent of each other as much as possible. Movies handle this problem with audio, but that's not an easy option yet for mathematics, nor is it easy to think about how to deal with it even if it were. Most of us are still restricted to making silent films.

- *A reader should not be compelled to shift attention constantly back and forth between text and figure.* Traditionally, vertex labels are used to coordinate text and figures, but what this means is precisely constant eye-shifting, trying to piece together different components of a diagram in order to follow reasoning. Bad psychology. Much better usually is a series of figures highlighting the component under consideration. Also, πS makes this easier since inserting TEX, hence verbal cues, into figures is relatively simple.

- *In drawing figures, think out how the material would be presented in spoken discourse.* Draw pictures that follow the same narrative, even if this means a fair amount of repetition. Computers can help in dealing with this sort of repetition.

- *Ask constantly if the figures really do convey the point they are meant to.* Redo them if necessary. Figures should be redrawn, as text is rewritten, until they are right.

- *Experiment.* Sometimes very small and subtle changes in a figure will have an enormous impact. Use imagination.

- *Don't attempt to say too much explicitly.* Good figures can be more suggestive than text. It is occasionally useful even to make them puzzling.

- *Keep in mind that people's interpretations of pictures vary unpredictably,* as thousands of psychological experiments show. Don't often depend entirely on pictures to make your point.

- *It is very rare for there to be too many illustrations in a mathematics paper.* Keep in mind that illustrations can serve a number of distinct purposes—for example, I find that I can often tell better what a paper with plentiful illustrations is about by skimming diagrams rather than by reading text. Pictures can often be read rapidly, and adding more should always be taken as a serious option.

Appendix 1. Setting up

In order to use πS you must have several programs installed on your computer:

- the programming language Python
- TEX
- a PostScript viewer
- a plain text editor
- the program πS itself
- PostScript versions of TEX fonts

The last two are part of the πS package itself. Once all the right files have been installed and certain environment variables have been set correctly, the process for producing a figure goes like this:

- you edit a Python program (simple text file) that uses the operations defined in the PiScript files to draw a figure or figures;
- you run Python on that file to produce a PostScript file (with the extension `.eps`);
- you view that PostScript file to see if all went as you meant it to, and if it did not you go back to the first step.

On most Linux systems all the necessary auxiliary programs and files are installed by default, so the only installation you'll have to make is of πS itself. So I'll start here by assuming that all but πS is installed on your machine, and tell you later about auxiliary programs required.

30. Installing πS under UNIX and its cousins

The home page for πS is

```
http://www.math.ubc.ca/~cass/piscript/docs/
```

The complete collection of Python modules you need, together with documentation and a directory full of examples, is in the file

```
http://www.math.ubc.ca/~cass/piscript/piscript-<version>.tar.gz
```

In addition you will probably want

```
http://www.math.ubc.ca/~cass/piscript/texfonts.tar.gz
```

Unpack the first. This will create and fill a directory `piscript-<version>` with a subdirectory `piscript`, which in turn contains subdirectories `examples`, `config`, and `docs`. The first contains—surprise!—examples of πS programs and the `.eps` files they output. The second contains certain sample configuration files, and the third, among other things, this manual.

There is another option, which I'll discuss in a moment, but in the simplest situation the next thing to do is unpack the `texfonts` file in the same directory `piscript-<version>/piscript`. This will give you a directory structure

```
piscript
    docs
    examples
    configs
    texfonts
        bluesky
        tfm
```

Of course, after unpacking anything you can discard the file you unpacked.

In most cases, your next moves are to (1) move the whole piscript directory where you want it; (2) register this location as part of your PYTHONPATH in your shell and then (3) choose a directory somewhere in your personal home directory to be your PYTHONCONFIGDIR, and register that also.

(If you are setting up πS for many users you will want to set it up in osme standard directory that everyone has access to.

How to do (2) depends on what shell program you are using (usually bash or tcsh in Linux). What you want to do is set the environment variable PYTHONPATH to the directory above piscript.

As for (3), what I have done is to create a 'hidden' directory .piscript in my home directory as well as a subdirectory configs in that, and then set PYTHONCONFIGDIR to $HOME/.piscript. I also make an empty file __init__.py in configs, which makes it a possible home for a Python package.

As an example of how to set environment variables, I myself use the shell program bash, and in the file .bashrc in my home directory I enter two lines:

```
export PISCRIPTCONFIGDIR='$HOME/.piscript'
export PYTHONPATH=$HOME/python/:$HOME/.piscript/
```

It happens that HOME/python is the directory above my πS directory piscript.

Normally, you should now be ready to write πS programs. As time goes on and you want to customize your environment, you can do so by adding files to directories in your Python path. But if you want to deviate from the standard setup, read on.

31. Alternative font access

Using the texfonts package that comes with πS is convenient, but very likely redundant. This section tells you what to do if this bothers you.

The TEX fonts take up a lot of space (about 18 MB) and if they are already somewhere on your machine you might want to use the ones already there. If you do this, you must locate the font files, and then change the first few lines of the file FontsConfig.py accordingly. You must also be sure to remove the texfonts directory in piscripts-<version> if you have unpacked the TEX fonts already. In any case, you can use system files only if the system font directories have a particular structure, but that is by now pretty standard, and should cause no trouble.

TEX needs two kind so files, the tfm (TEX font metric) files and the PostScript Type 1 font files for the standard TEX distribution. As long as you have TEX on your system, the tfm files are definitely on your machine somewhere. You can find them by looking for a specific one, say cmr10.tfm. On a UNIX machine, you can use find to locate them:

```
[cass@newhaven ~]$ find / -print | grep cmr10.tfm
find:  /usr/libexec/utempter:  Permission denied
...
/usr/share/texmf/fonts/tfm/public/cm/cmr10.tfm
...
```

This tells me that the tfm directory is /usr/share/texmf/fonts/tfm/. Put this information in the file piscript/FontsConfig.py. Second are the .pfb files, which specify the actual shapes of characters. So you look for cmr10.pfb:

```
[cass@newhaven piscript]$ find / -print | grep cmr10.pfb
find:  /usr/libexec/utempter:  Permission denied
...
/usr/share/texmf/fonts/type1/bluesky/cm/cmr10.pfb
```

```
...
```

The directory you are looking for is `/usr/share/texmf/fonts/type1/bluesky`, which you also put in `piscript/FontsConfig.py`.

32. TEX environment

The next thing you must do before final installation is to set the global TEX environment in the file `TexConfig.py`, which ought to be placed in the `piscript` directory. Most users, especially at first, should just copy one of the files `configs/PlainTexConfig.py` or `LaTexConfig.py` into `piscript/TexConfig.py`. In time, you will probably want to customize the TEX environment. How to do this is explained in the main text.

33. Installing under Windows

The simplest to do is run

```
python setup.py install
```

inside the `piscript` directory. It will put the πS package in a system-wide standard location. If you want to customize, reading over what I say about UNIX systems should be enough.

34. Auxiliary programs

I discuss here the auxiliary programs you'll need in order to get things working.

Python

The first requirement for using πS is that your computer have Python installed. I cannot tell you how to achieve that if it is not already done, but I can tell you that the home page for Python is

```
http://www.python.org/
```

Depending on the operating system of the machine you are using, running Python on a file varies. The simplest will work on most systems—just click on the Python file. In order for this to work, your system must be configured to run Python on `.py` files. On Windows machines, doing this will have a disconcerting effect - a DOS window will pop up and disappear almost immediately. Not so good for keeping track of your program.

A second method is to run Python on the file in a terminal window:

```
python x.py
```

A third is to run the file itself. On a UNIX-like machine, you must have something like `#!/usr/bin/python` in the first line of your file, and the file must be executable. Then you can type `./x.py` in a terminal window. On a recent Windows machine, just type `x.py` in a DOS window.

A fourth is the one I use, which I recommend strongly, and that to use the utility `make` to manage your figure files. This deserves a separate and later discussion.

TEX

You must have TEX installed, and you must have access to certain font files. Un UNIX-like machines, TEX is usually installed by default, and causes no problems. For Windows machines, you must have a version of TEX that can be run by a terminal command-line like `tex` or `latex`. The program MiKTeX at

```
http://miktex.org/
```

is available without cost and does fine.

In order to use TEX in figures, πS must know how to locate certain kinds of font files. First are the `.tfm` (TEX font metric) files, which specify the sizes of characters. Second are the `.pfb` files in which the actual character descriptions are to be found. I'll explain more about this later, after I have explained how to install πS.

PostScript viewer

In order to see your PostScript figures, you'll need a PostScript viewer, such as the combination of `GhostScript` and `gv`. It is `GhostScript` that does the basic interpretation of PostScript files, but normally you would use a more convenient viewing program to allow easier interaction with it.

By far the best PostScript viewer is `gv`, which is now maintained as part of the `GNU` project. But as far as I can see it is not available on Windows machines, and perhaps a bit tricky to install on Macs. Alternatives are `GSView` and `GhostView`. A useful general source of information, with good links, is

```
http://en.wikipedia.org/wiki/Ghostscript
```

More specifically, for the most recent version of `gv`:

```
http://www.gnu.org/software/gv/
```

Appendix 2. A brief introduction to Python

Python is an elegant programming language. Although it does not produce programs that execute as rapidly as those produced by Java or C++, it is very easy to write simple programs quickly, and when it comes to needing more speed it is not hard to interface to programs written in other languages. It has been around for more than a dozen years, but seems to have become popular only recently. It deserves its new-found popularity.

My own purpose in learning Python was to design a graphics tool to replace the programming in PostScript that I have been doing for many years. I believe that I have accomplished this, with what I call the PiScript modules. But since then I now use Python to write all kinds of simple programs in. For example, although I will not explain it here, the regular expression package for Python is far more convenient than than the one for PERL, and in fact I see no reason for anyone at any time from now on to write a program in PERL. Thank Heaven.

There are lots of sites in the Internet where one can find an endless amount of information about Python, so I am not going to write a full blown text here. What I am hoping is that this note will get you to the point where you can write simple programs and then be able to look around for information on how to write more advanced ones. I hope in particular that this will be just enough so that you you read my sample PiScript programs and then make your own. In learning Python, I myself used the book **Learning Python** written by Mark Lutz and published by O'Reilly, but gosh! it is awfully—and painfully—verbose.

Documents available on the Internet include

`http://docs.python.org/tut/tut.html` (official tutorial)

and a plethora of stuff listed at

`http://wiki.python.org/moin/BeginnersGuide/Programmers`

35. Starting

I believe in teaching programming by examples. Here is a program that calculates and displays the sum of the first 100 integers:

```
#!/usr/bin/python

# calculates sum of first 100 integers
s = 0
for i in range(1, 100):
    s += i
print "s = " , s
```

The name of the file it is in is `sum.py`.

It's pretty simple. Let's go through it line by line.

```
#!/usr/bin/python
```

This first line is for systems running some flavour of UNIX (such as Linux or MacOS). It is not necessary, but allows you to set up the file as an executable on these systems by setting its permissions flag to be 755. Otherwise, the normal way to execute the program is to type `python` plus the file name on a command line: `python sum.py`. It is not necessary that the file have extension `.py`, but it is a good idea.

```
# calculates sum of first 100 integers
```

This is a comment. The sign # makes the remainder of the line a comment. Longer multi-line comments can be enclosed between a matching pair of triple quotation marks `"""`.

```
s = 0
```

Like most other languages these days, `=` is variable assignment. One mildly eccentric but lovable feature of Python is that the types of variables don't have to be declared. Python knows here that `s` is an integer variable, and keeps track of that fact. This **dynamic typing** might occasionally cause trouble, but not often. Python will complain about a variable's type only when it is asked to perform some task with the variable that its type is not equipped for.

Statements may be ended with a semi-colon or not. Separate statements on a single line must be separated by semi-colons.

```
for i in range(1, 100):
```

This is one of the two common loops in Python. As before, `i` is not declared, and in fact its type is somewhat indeterminate in a `for` loop. The `range(1,100)` is actually Python list (basically, an array) of the integers 1, 2, ... , 99. You can see this by

```
print range(10)
```

What the loop does is just run through the array, setting `i` to be each element it encounters in turn.

```
    s += i
```

The only really eccentric feature of Python is the way it **requires** indentation by one or more tab spaces to mark blocks of code that are marked with `{ ... }` in C or Java. Even comments. Indentation has to agree with the code environment. The lines that require subsequent indentation end in a colon. These include function definitions, loop beginnings, and conditionals.

Unlike in C or Java, `++` and `--` are not part of the language.

```
print "s = " , s
```

The command `print` works pretty much in the predictable way. Output is to standard output. Normally `print` output is followed by a carriage return, but the comma annuls that. Putting commas in the middle of `print` statements just concatenates output with a little extra space but no carriage returns.

It doesn't show up in this short program, but there are two other characteristic features of Python. The first is that, unlike C or C++ but just like Java, you do not have to allocate memory for objects. This is both good and bad news, since clever memory handling is a major component of fast programs. Still, it is very, very convenient.

36. Data types

The built-in data types of Python are

integers
floating point numbers
Boolean
strings
lists
dictionaries
files
functions
classes

Integers. These are different in Python from what they are in most programming languages, in that they are of arbitrary size. In other words, whereas in Java or C the maximum ordinary integer is in the range $[-2^{31}, 2^{31} - 1]$, in Python the transition from 2^{31} to $2^{31} + 1$ is done correctly and silently. There is a cost to this amenability, since

in most machines the CPU handles integers modulo 2^{32}, and in some modulo 2^{64}. Therefore, as soon as large integers are encountered in Python they will take up more than one machine word, and dealing with them will be correspondingly slow.

Integer division gives the integral quotient. Thus $6/5$ returns 1. One wonderful feature of Python for a mathematician is that integer division returns the true integral quotient. So `n/m` is always the integer q such that $qm \leq n < (q+1)n$, and similarly `n % m` (i. e. n modulo m) always lies in $[0, m)$. So `-6/5` is -2 and `-6 % 5` is 4.

Floating point numbers. Nothing special, except that there seem to be no equivalent in Python of single precision floating point numbers (floats in Java). All are double precision. I do not know to what extent Python floating point numbers depend on the machine at hand, or whether the compiler takes the IEEE specifications into account.

In displaying these numbers, the default is to reproduce the full double precision expression. This often annoying behaviour can be modified by using a format string, as in C. Thus

```
print "%f" % 3.14159265358979
```

will produce `3.141593`. Other options allow more control. See

`http://www.webdotdev.com/nvd/articles-reviews/python/numerical-programming-in-python-938-139_3.html`
`http://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks`

for more information on formatting of floats. (The second site has a useful table around the middle.)

Because division of integers yields the integer quotient, you must be extremely careful when dividing variables that might be either integers or floating numbers. In these circumstances, it is safest to convert either numerator or denominator to a float first, say by multiplying one or the other by 1.0. *Or by using the operator* `float`, *which changes* x *to a float.*

Thus `float(1)` returns 1.0 When you do this, be careful to group terms with parentheses. $x = 1.0 * a/b$ (incorrect) is not the same as $(1.0 * a)/b$ (correct).

Strings. To build strings by concatenation of data types other than strings, you must use the `str` function. Thus

```
print "x = " + str(x)
```

But here,

```
print "x = ", x
```

will also work or even

```
print "x = ",
print x
```

There are functions to turn strings back into other types, too. Thus `int("3")` returns 3 and `float("3.3")` returns 3.3. These are useful in parsing command lines (discussed in a later section).

Booleans. These are `True` and `False`. Boolean operations are English words: `not`, `or`, `and`, `xor`, ...

Lists. A list is essentially an array. It is different from other types of array in Python that I never use, in that you can change its entries, and you can enlarge it or shrink it. Any sequence of items can be put into a list—lists can be arrays of objects of very different types. Lists are normally grown in steps. Thus to get a list `a = [0,1,2,3,4, ..., n-1]` one writes

```
a = []
for i in range(n):
    a.append(i)
```

but if the list is fixed ahead of time you can just write `a = [0,1,2,3,4]`. You build lists by appending data to it, and you can delete entries from it. The most useful deletion method is `pop`, which removes and returns the last item in the list. This allows you to simulate stacks in Python very easily.

Dictionaries. A dictionary is a special kind of list, one of of keys and values. It is the analogue of hash tables in other languages, but in Python is one of the basic types of data. Here is a sample dictionary:

```
d= {"red":[1,0,0], "green":[0,1,0], "blue":[0,0,1]}
```

You access the values for a given key very conveniently: `r = d["red"]`,a nd you can add entries to the dictionary equally conveniently: `d["orange"] = [1,0.5,0]`. You can check if a key is in the dictionary: `if d.has_key("red"): ...` , and in fact you should do that if you are not sure whether or not a key exists, because referring to `d[k]` when `d` does not have `k` as a key is an error. You can list all keys: `k = d.keys()`

Functions. A function is defined like this:

```
def sum(m, n):
    s = 0
    for i in range(m, n):
        s += i
    return s
```

If you are calling a function with no arguments, be sure to use parentheses, because functions are also objects. They can be passed as arguments. Thus

```
def newton(f, x):
    for i in range(10):
        fx = f(x)
        x -= fx[0]/fx[1]
    return(x)

def sqr(x):
    return([(x*x-2.0), 2.0*x])

print newton(sqr,1)
```

is OK. (Note how I have made `sqr` return floating point numbers.) Using functions as variables makes up to some extent for the lack of a `switch` or `case` in Python, which is for some of us a major and mysterious lack. You can find lots of puzzled complaints about this on the 'Net.

If you want to change the value of a global variable inside a function you must declare it to be global inside the function. Thus

```
def set(n):
    global a
    a = n
```

not

```
def set(n):
    a = n
```

Functions return the `nil` object by default.

Files. The basic procedures involved in using files for input and output are very simple.

```
f = open("output.txt", "w")
f.write("abc\n")
```

and

```
f = open("output.txt", "r")
s = f.read()
```

and

```
f = open("output.txt", "r")
s = f.readline()
while (s):
    s = f.readline()
```

But there are some things to watch out for. The most important is that when you are through using your file, you must close it. For one thing, files definitely don't flush their output until `flush()` or `close()` is called. For another, at least in Windows, you will not be able to do anything else with the file, such as remove it, until it is closed. Also more important in a Windows environment is that a binary file should be handled with a tag "b", as in "rb" or "wb", since otherwise Windows interprets some characters, such as EOF, specially.

Classes. As in other object oriented programming languages, you can create new data types. Here is one that defines a stack object:

```
class Stack:
    def __init__(self):
        self.list = []

    def push(self, x):
        self.list.append(x)

    def pop(self):
        return self.list.pop()

    def toString(s):
        return str(s.list)
    toString = staticmethod(toString)

# --- sample usage -----------

s = Stack()
s.push(0)
print Stack.toString(s)
n = s.pop()
```

This isn't a very useful class, since it does nothing more than lists, but if you want to feel at home in Python, maybe you'll use it. As you can see from this, Python can be rather `self`ish. This gets a bit annoying, I have to say. It is the analogue of `this` in Java, but whereas it is implicit in Java it is screaming right in your face in Python. For better or worse. When your program has `s.pop()`, this translates to `Stack.pop(s)` in the class—i. e. the calling instance becomes the first argument. One pleasant feature of classes is that you can overload certain operators like + and −, but I won't explain that here.

37. Variables

The most fascinating thing about variables in Python is that their types are dynamically determined. You should think of every thing in the program having a type that gets stuck to it transmitted in assignments. Variable types can vary. Thus

```
x = 3
x = "Hi, there!"
```

is an entirely legal successive pair of lines.

Also, as in Java, memory allocation is handled automatically by garbage collection. This is both good and bad news, since hideous efficiency in C++ programs often relies on clever memory allocation.

38. Loops

I use just two kinds, `for` and `while` loops. These are especially useful when combined with `break` (which exits the current loop) and `continue` (which goes back to the beginning of the loop) inside the loops. The basic `for` loop is standard:

```
for i in range(3, 17):
    ...
```

but `range` denotes here the array of numbers `[3, ... , 16]` and can be replaced by any kind of sequence.

39. Conditionals

```
if ... :
    ...
elif ... :
    ...
else:
    ...
```

Testing equality is done with `==`.

40. Modules

There are many packages in Python which may be optionally loaded. One of the most important is the `math` module, which is used in a program like this:

```
import math
...
c = math.sqrt(a*a + b*b)
...
C = 2*math.pi*r
```

or

```
from math import *
...
c = sqrt(a*a + b*b)
```

```
C = 2*pi*r
```

Importing basically makes names of variables and functions available. You have to be careful—in the last example, the `sqrt` from `math` will replace whatever `sqrt` function you have defined locally.

You can also make your own modules, which makes for much less redundant programming.

41. Command line arguments

If the file `sum` is

```
#!/usr/bin/python

import sys
m = int(sys.argv[1])
n = int(sys.argv[2])

# calculates sum of integers from m up to n
s = 0
for i in range(m, n):
    s += i
print "s = " , s
```

then

```
sum 1 10
```

will print out the correct sum. Also, unlike in some other languages, `argv[0]` is the command itself. Thus in

```
python sum.py 5 10
```

the 0-th argument is `sum.py`.

42. Learning more

The following list of web sites devoted to quirks of Python was brought to my attention by Christophe K.:

```
http://jaynes.colorado.edu/PythonIdioms.html
http://zephyrfalcon.org/labs/python_pitfalls.html
http://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks
http://www.onlamp.com/pub/a/python/2004/02/05/learn_python.html
http://www.ferg.org/projects/python_gotchas.html
```

Appendix 3. Inserting your beautiful figures into TEX files

Including your figures in a TEX file is easy.

Latex. If you have produced a figure `x.eps`, you could insert it in a (rather silly) LATEX file like this:

```
\documentclass[a4paper,12pt]{article}

\usepackage{epsf}
\begin{document}

Here is the figure {\bf x.eps}:

\epsfbox{x.eps}

\end{document}
```

There are other ways to do this in LATEX, but this is the simplest and most flexible that I know of.

Plain TEX. For plain TEX it is even easier—just put `\input epsf` at the head of your file. Like this:

```
\input{epsf}

Here is the figure {\bf x.eps}:

\epsfbox{x.eps}

\bye
```

Both. There is one additional useful thing to keep in mind—you can modify the size of the figure inserted by specifying width or height, as in one of these variants:

```
\epsfxsize=3cm
\epsfbox{x.eps}

\epsfxsize=3truecm
\epsfbox{x.eps}

\epsfysize=1truein
\epsfbox{x.eps}
```

It is important to know that the effect of one of these size specifications disappears after each use. Another useful variation is this, which centers the figure:

```
\leavevmode
\hfill
\epsfbox{../examples/int.eps}
\hfill
\null
```

At any rate, run TEX on the tex file to get a `.dvi` file; run `dvips` on the `.dvi` file to get a `.ps` file, and `epstopdf` to get a `.pdf` file. If the name of the TEX file is `fig.tex`, for example, this goes

```
tex fig
dvips -o fig.ps fig
epstopdf fig.ps
```

For more information, look up information on `dvips` on the Internet.

Appendix 4. The make utility

I recall my indubitably excellent advice—be sure to name your `.py` and the corresponding `.eps` files with the same prefix, so that `x.py` produces `x.eps`. This will be a definite step towards maintaining sanity. We'll see some additional very good reasons to do that in this section.

If you have followed the conventions I recommend without qualification, the file `x.eps` will depend on the file `x.py`. When you change `x.py`, you have to run Python on it to get the corrected file `x.eps`. You can automate the update process, by using the program `make` to keep track of the dependency.

The simplest way to do this is to keep in the same directory as your `.py` files a file named `makefile`. In this file, you first specify that a file with extension `.eps` depends on one with extension `.py`. You do this by putting at the very top of `makefile`

```
.SUFFIXES: .py .eps

.py.eps:
    python $*.py
```

The term `$*` here stands for the prefix of a file. The effect of this is that if you type

```
make x.eps
```

in the directory, then the system will look for a file `x.py`, check to see whether it has been modified since `x.eps` was last modified, and if so run Python on it. I call that very clever. But you can carry this one step further, by putting in `makefile` a list of files of interest. So in this case we put lower down the line

```
all:  x.eps y.eps
```

and now when you type just `make` the system will update `x.eps` and `y.eps` if necessary. At least as long as you have followed the conventions I suggest and name your output file consistently with the `.py` file in the `init` line.

This can be carried much further. For example, `.dvi` files depend on `.tex` files, etc. But there is one observation useful right here. If you want to produce `.pdf` files instead of `.eps` files, your local `makefile` should look like this:

```
.SUFFIXES: .py .eps .ps .pdf .dvi .tex

.tex.dvi:
    latex $*.tex

.dvi.ps:
    dvips -o $*.ps $*.dvi

.py.eps:
    python $*.py

.eps.pdf:
    epstopdf $*.eps

.ps.pdf:
    epstopdf $*.ps

# ------------------------------

all:  x.pdf y.pdf
```

Appendix 5. Commands in the Vector class

These operations must be preceded by "Vector" plus a period, as in `Vector.sum`.

Vectors and points are always arrays $[x, y, \ldots]$.

- `sum(u, v)`

Returns $u + v$. Vectors here, as in all the vector functions are of arbitrary dimension unless specified otherwise. The way this and other vector operations is used is like this:

```
import piscript.Vector

...

u = [0,1]
v = [2,-1]
w = Vector.sum(u,v)
```

- `diff(u, v)`

Returns $u - v$.

- `scale(u, c)`

Returns cu, where c is a number.

- `length(u)`

Returns $\|u\|$.

- `arg(u)`

Returns the angle between u and the x-axis. The vector u must be 2D.

- `interpolate(P, Q, t)`

Returns $(1 - t)P + tQ$.

- `linethrough(P, Q)`

Returns $[A, B, C]$ with P and Q on the line $Ax + By + C = 0$. Here P and Q must be 2D.

- `prod(a, b)`

Returns either the dot-product, the matrix-vector product, or the matrix-matrix product. A matrix is an array of its rows.

- `dotproduct(u, v)`

In any dimension.

- `crossproduct(u, v)`

Only for 3D vectors.

- `rotate(u, A)`

Returns u rotated by A.

Appendix 6. Index of commands

Commands are followed by the number of the page on which they are described.

Basic commands

Vector utilities

3D commands