7:47 a.m. June 23, 2010

PiScript—a drawing tool for mathematicians

by Bill Casselman

Preface

Producing good mathematical illustrations is a major part of good mathematical exposition. Computers have made this a totally different experience from what it used to be, but it is still not generally recognized as a simple task. I hope to change that with the program PiScript (ΠS), which this manual introduces.

ΠS is essentially 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, especially text produced by TeX, is straightforward. In ideal circumstances, you can reproduce your entire normal TeX environment in ΠS.

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 both simple figures as well as more complicated ones of extremely 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 flexible 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, most of them have some trouble integrating text with graphics conveniently. One notable exception is PyX, but it is a very, very unwieldy tool otherwise.

One solution to this problem is the one I 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 several major parts:

Contents

1. Drawing in 2D
2. Text in figures I. TeX
3. Text in figures II. PostScript
4. Paths
5. 3D drawing
6. Miscellaneous
7. Coordinate systems
8. Advice on illustrating mathematics

The last part is no more than an outline, and will be expanded in the future. In addition:

Appendices

A1. Setting up
A2. A (very) brief introduction to Python
A3. Inserting your beautiful figures into TeX files
A4. The make utility
A5. 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; and Christophe K. for helping me set up πS under Windows. I'd like to thank Günther Ziegler for arranging visits to Berlin in order to work on πS, and a few long-suffering guinea pigs . . . er, I meant to say students . . . at the Berlin Mathematical School at TU-Berlin for helping me chase out bugs and add features. Finally, I'd like to thank David Maxwell for taking a big hand in improving the TeX and other font facilities. As far as text handling is concerned, and because of other valuable suggestions made by him for improvement, he should be considered a partner in this project.

Part 1. Drawing in 2D

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

1. The graphics model

The graphics model of πS is essentially that of PostScript, and in fact its principal output is PostScript code that utilizes the PostScript graphics environment. PostScript is a complete programming language, but it was not intended for human use. It was designed primarily 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—changed into something a computer could deal with directly—before interpretation. On the contrary, PostScript is expressed in RPN (**R**everse **P**olish **N**otation) format, which allows commands to have immediate effect. (RPN was invented for the most pure of reasons by the most pure of 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 then removed and operated on, as soon as possible, when operators appear. 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				

Expressions do not have to be put on hold until they have been completely read—as soon as an operator is encountered, it is applied. 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 PostScript 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, the human mind is surely based on the alternate paradigm of recursion and context-free grammar. With the more conventional Python interface, that need not bother us.

The graphics model of PostScript is fairly simple. First 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 in πS that do some bit-map manipulation). This is often called **vector** or **scalable** graphics. The principal task that πS performs is to construct and draw paths. Even setting text is essentially a matter of drawing paths. Once a path has been constructed, one can fill its interior with color or merely stroke its outline. The paths are constructed in a certain coordinate system, which the programmer can change as he or she goes along. I repeat: paths are first constructed, then drawn. When then they are actually drawn, certain parameters (such as color) are applied.

𝒫𝒮 and PostScript both use 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.

It is by no means necessary to understand PostScript in order to use 𝒫𝒮, but it will help in understanding some of the decisions made in developing it. For a brief account, the Wikipedia page

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

is instructive.

I now run through a description of the basic commands available in 𝒫𝒮. Many of these 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

Once all the right files have been installed and certain environment variables have been set correctly (see Appendix A.1), 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, if everything goes well, to produce a PostScript file (by default, 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.

The text file should normally have extension `.py`.

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

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

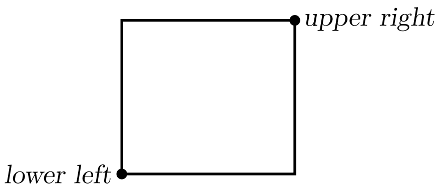
This automatically imports as well the Python `math` package as well as the 𝒫𝒮 package `Vectors` and the class Vector it contains.

In any 𝒫𝒮 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).

● `init(w, h)`
`init(llx, lly, urx, ury)`
`init(..., "noclip")`

There are several options to `init`, and I shall explain more in a moment. To preserve your sanity, your 𝒫𝒮 program files should always end with the extension `.py`, and the prefix of the `.py` file should match that of the PostScript output. This advice is reinforced by the simplest option for `init`, in which by default the output file for the program `xxx.py` is set automatically to `xxx.eps`.

The numerical arguments in these options set the bounding box of your figure. the corresponding PostScript figure will have bounding box `(0 0 w h)` or `(llx, lly, urx, ury)`. 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). 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. The principal point of using the longer form, with a lower left corner other than $(0, 0)$, is to get around the fact that printers often refuse to print along the margins of a page.

The most general form of the arguments of `init` is `(<output>, <bounding box>, <clipping option>)`. As we have seen, the output specification can be blank. But if you don't want your program `xxx.py` to produce `xxx.eps` you'll have to specify the output. It must be a string, like `"xxx"`, surrounded by double quotes. Here are some examples.

Output argument	Output file	
`".ps"`	`xxx.ps`	(where the program file is `xxx.py`)
`"yyy"`	`yyy.eps`	(even if the program file is `xxx.py`)
`"xxx.ps"`	`xxx.ps`	

There is a subtle difference between the two possible extensions `.eps` and `.ps`, but all you have to know is that if you are producing a PostScript file with several pages:

If you are making several pages in one program then (a) you must choose ".ps" output and (b) you should set the lower left of your bounding box to be $(0, 0)$.

The reason for the second rule is that many programs that process your PostScript file, such as printers or PDF renderers, forget the bounding box specification on the second and subsequent pages and therefore introduce an unwanted offset.

Unless the argument `"noclip"` is used, all subsequent drawing is restricted to within the current bounding box. If `"noclip"` is the last parameter, the figure will be allowed to overflow its bounding box.

- `beginpage()`

Begins a new page. The important point about this is that you can output files of several pages, although usually you will want to output only one. Pages are isolated from each other—by default, all changes in the graphics state are entirely restricted to one page, so that each page may be accessed independently. 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 beginning by translating the origin to (llx, lly). Also, `beginpage()` causes the default graphics state to be saved on the graphics stack, and a new copy pushed above it. (Exactly what this means will be explained later.)

- `endpage()`

Ends a page, restores the default graphics state so the next page starts 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 that violate the page structure.

- `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()`.

The minimal πS program is thus something like

```
from piscript.PiModule import *

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

It opens a file called `something-or-another.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 (but it should have a ".ps" extension):

```
from piscript.PiModule import *

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

As a program proceeds, its coordinate system may change, and as this happens the coordinates of its corners will change as well. The real, physical limits of your figure—its bounding box—are set once and for all in the initialization. It may not be changed dynamically, but it is possible to see what it is in terms of current coordinates.

- `currentbbox()`

Returns the current bounding box in terms of current coordinates. This is a polygon, the array of its corners. We can also recover the width and height of our figure:

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

Of course these are completely determined by the bounding box, which is statically determined, but if you change the bounding box in `init` you might like to avoid having to change the references to fixed numbers all through your program. For example, in the opening coordinate system to the point 10 points left of and 10 points down from the upper right hand corner is `(width()-10,height()-10)`.

3. Confession

I have not in fact told the entire truth. In excuse, I quote the immortal words of Don Knuth, who tells us in the preface to the **TEXBook**:

> *Another noteworthy characteristic of this manual is that it doesn't always tell the truth. . . . The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions.*

I'm not sure I agree with this completely, because a friend of mine who is a child psychologist once told me that the opposite is true of children, in the sense that they often have great trouble readjusting something they once believe. But in the case at hand I feel justified. How did I lie? *It is in fact not necessary to use* `beginpage`/`endpage` *if you are only doing one page.* So the minimal πS page outline is in fact more like

```
from piscript.PiModule import *
init(100, 100)
...
<drawing in here>
```

```
...
finish()
```

Everything here *is* really necessary. However, using `beginpage/endpage` does no harm.

If you ask why I have not started out by telling you the truth, I have to confess it is because a number of tricky questions arise that I don't want to deal with right now. For example, suppose you do some graphics between `init` and `beginpage`. Is it then lost? The safest thing to do is this: if you use `beginpage/endpage` then you should do graphics only between these two commands.

4. Simple drawing commands

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([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 forms of argument are especially useful (here, and also in other commands where they are acceptable) when feeding in points calculated by some other routine.

From now on, a P or a V listed as an argument for a command will allow as well either (a) a list of numbers (such as (x, y) above) or (b) a single object P which is an array in the sense that (i) entries $P[i]$ are defined and evaluate as numbers and (ii) `len(P)` *is the number of items in the array.*

Useful examples of such arrays are Python lists `[...]` and tuples `(...)` as well as Vectors, which are part of the πS package. I recall that tuples in Python are immutable, which is to say they cannot be changed in any way at all, whereas lists are extremely changeable.

In this manual, I'll write `(...)` *when any array in this sense is acceptable as an argument, and* `[...]` *when a mutable array is required.*

There is going to be some mild confusion, though, for mathematical reasons that will be apparent later on.

- `lineto(P)`

Adds to the current path the line from the current point to P. 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 single path from 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))`

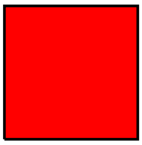
The commands described earlier tell you how to construct a path, but they do not display it. There are two ways to do so. The first of these is `stroke`. It draws along the current path in gray scale g, or color (r, g, b). If no arguments are given, it strokes in the current color. At the start of every page, this color 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 a file somewhere and import their definitions. One can also use the arrays to manipulate colors, for example by interpolating them or darkening them.

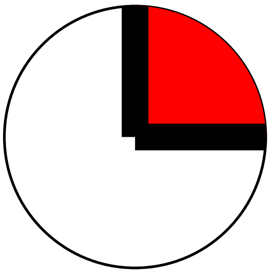
- `fill()`
  `fill(g)`
  `fill(r, g, b)`
  `fill((r, g, b))`

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 seems to draw a perfectly fine square, but if we zoom into the starting corner we can see that it is flawed:



The other corners don't have this problem. We'll see later (in the discussion of `closepath`) how to fix it.

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 Sat Jul 11 22:20:35 2009
%%BeginProcset:
/ReEncodeFont { exch findfont << >> copy dup 3 2 roll
/Encoding exch put definefont } def
%%EndProcset
%%BeginProlog
%%EndProlog

%%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. As I said earlier, PostScript was not designed primarily with human readability in mind. 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.

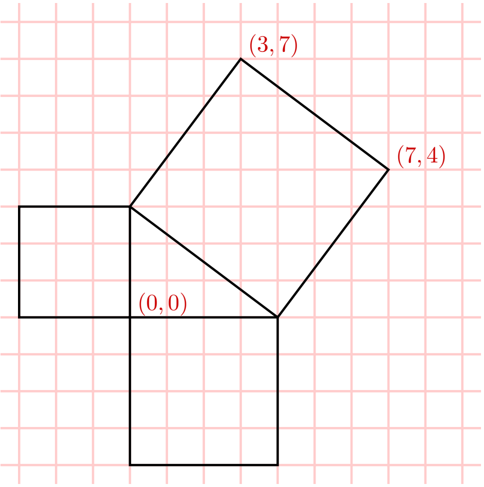
Both πS and PostScript always lay a figure over what has been previously drawn. There is no way to achieve partial visibility by adjusting a transparency factor. (This may be fixed in future versions of πS that are capable of producing output other than PostScript.)

Incidentally, πS 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.

5. A simple example

With the commands I have mentioned, you can already do some interesting drawing. After a quick hand-sketch, you can produce a familiar figure (as I'll do often, I have added a coordinate grid that is not accounted for in the code):

```
newpath()
moveto(0,0)
lineto(4,0)
lineto(4,-4)
lineto(0,-4)
lineto(0,0)
stroke()

newpath()
moveto(0,0)
lineto(0,3)
lineto(-3,3)
lineto(-3,0)
lineto(0,0)
stroke()

newpath()
moveto(0,3)
lineto(4,0)
lineto(7,4)
lineto(3,7)
lineto(0,3)
stroke()
```



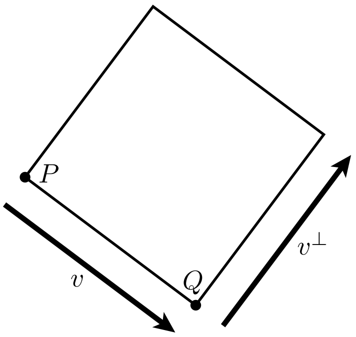
There are many reasons not to be so happy with this. Suppose you want to draw some other Pythagoras configuration? You'd have a lot of numbers to change. Also it seems somewhat redundant to put in all those `linetos` to make the squares, when you're really doing the same task over and over—making a square.

For the first problem, we can use variables:

```
a = 3
b = 4
c = math.sqrt(a*a+b*b)
```

and change the later code accordingly. For the second, we can define a procedure `mksquare` that constructs a positively oriented square with a given side from P to Q.

```
def mksquare(P, Q):
    P = Vector(P)
    Q = Vector(Q)
    v = Q - P
    vperp = v.perp()
    moveto(P)
    lineto(Q)
    lineto(Q + vperp)
    lineto(P + vperp)
    lineto(P)
```



For ease of reading this uses Vectors, which will be explained later. This makes possible a kind of vector algebra, as you can see. Given this procedure, making the Pythagoras figure looks pretty simple:

```
newpath()
mksquare([0,0], [0,a])
stroke()

newpath()
mksquare([0,0], [0,-b])
stroke()

newpath()
mksquare([0,a], [b,0])
stroke()
```

6. More about drawing

At any given moment, the graphics in PostScript or πS is done in a particular graphics environment or **graphics state**, which the programmer can change every now and then. Part of the graphics state 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. The paths are the important part of your program, and it is important that the part of your program that draws paths be readable.

I have already introduced a few basic drawing commands. In some sense, they are just about all that's really necessary. But it is convenient to have some more at hand.

- `rmoveto(V)`

Shifts the current point by the vector $V = [dx, dy]$, 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 both realized as arrays. 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 array $[dx, dy]$ 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 not position).

- `rlineto(V)`

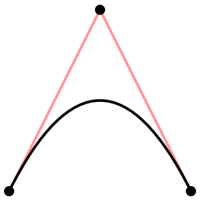
Adds a line segment to the current path, with end point relative to the current point. The `r` in `rmoveto` and `rlineto` stands for “relative”.

- `quadto(P1, P2)`
- `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 πS 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 P_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.

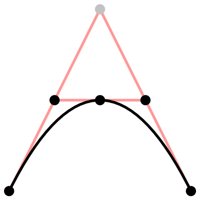
```
newpath()
moveto(0,0)
quadto([10,20],[20,0])
stroke()
```

produces (with control points and tangent lines also drawn):



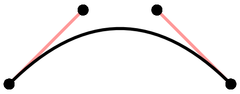
This simple geometrical relation makes quadratic Bézier curves the natural choice in many situations. One very natural one is in constructing contours of a function $f(x, y)$ whose gradient ∇f is known.

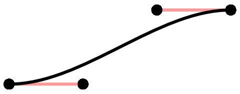
These and the cubic curves are drawn very efficiently by a computer, because of how they behave under subdivision. If a quadratic Bézier curve is divided in equal halves, each half is again a quadratic Bézier curve whose control points are simple to construct. The following figure illustrates what happens:



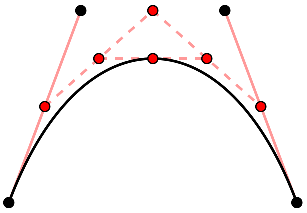
- `curveto(P1, P2, P3)`
- `rcurveto(V1, V2, V3)`

Adds to the current path the cubic Bézier curve with control points P_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.





As with quadratic Bézier curves, cubic ones are also constructed by a computer through repeated subdivision into halves. The following figure shows that (a) each half of a cubic Bézier curve is itself a cubic Bézier curve, and (b) the control points of those halves may be found by successive linear bisection. Combining these, we can see that the whole curve can be assembled from small line segments constructed by bisection, which is very efficient on a computer.



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

$$\begin{aligned} P_1 &= P_0 + (1/3) f'(t)\Delta t \\ P_2 &= P_3 - (1/3) f'(t + \Delta t)\Delta t\,. \end{aligned}$$

Because of this, Bézier curves can be used to approximate any graph whose slopes are specified at given points, or to draw a path whose velocity is specified at given points—for example, for plotting trajectories of solutions of differential equations, and in particular of integrals.

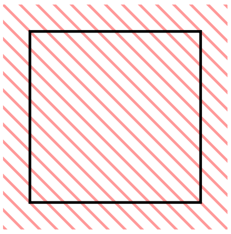
There is a third thing you can do with a path besides stroke or fill it.

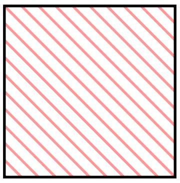
- `clip()`

This clips (i. e. restricts) 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 clipping path is part of the graphics state, since the effects of this command can be controlled with the `gsave/grestore` commands explained later on.

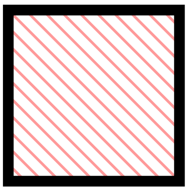
The only difference between the following pictures is the added three lines that clip to the box.

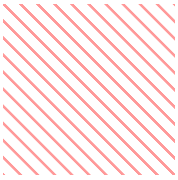
```
gsave()
newpath()
box(2,2)
clip()

newpath()
for i in range(N):
    moveto(-10, 10)
    lineto( 10, -10)
    translate(dx, dx)
stroke(1,0.6,0.6)
grestore()

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





The clipping restriction should not be in place when the box is stroked, because the clipping will affect the stroke more than you could guess. The use of `gsave/grestore` is crucial here. In the figures below, the effects have been exaggerated by thickening lines. On the right the clipping has been left in effect when stroking is done. Not so yummy.





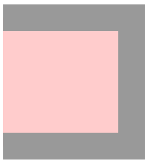
As I have already mentioned, unless `init` has a final “noclip" argument, every πS figure is clipped to its bounding box.

- `closepath()`

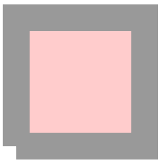
Closes up the current path to the location of the last `moveto`. Even if the last point you draw to is the same as the first point you moved to, the path will not be in fact closed unless you use this operation—the beginning and end points will be treated differently from the other vertices of the path. 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()`.

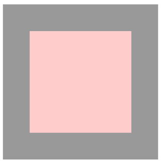
Also, `fill` automatically closes paths before filling. Thus we have the following three figures (with thickened line widths to exaggerate effects):



```
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)
```

7. Familiar shapes

This section introduces you to a number of simple shapes constructed by a sequence of primitive commands. Some of them involve angles.

- `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°. 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 $90 \cdot 180/\pi$.

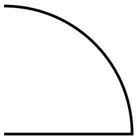
- `arc(P, r, A, B)`

Adds to the current path an arc in the positive direction from A to B, centred at P 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.*

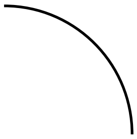
Thus

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



is different from

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



The reason for this seemingly bizarre behaviour is that it enables you to construct continuous curves from separate arcs rather easily. Without this behaviour you'd have to do some calculation with $\sin$ and $\cos$.

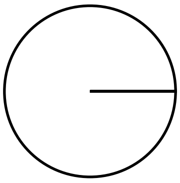
- `arcn(P, r, A, B)`

Goes in the negative direction.

- `circle(r)`
  `circle(P, r)`

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

```
newpath()
moveto(0,0)
circle(1)
stroke()
```

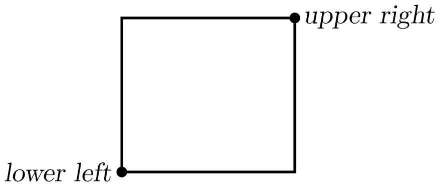


- `box(w, h)`
  `box(P, V)`

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 P, upper right at $P + V$.

- `boundedbox(llx,lly,urx,ury)`
  `boundedbox(P1, P2)`

This makes a *bounded box* with lower left corner at P_1 and upper right corner at P_2. It is especially convenient for use with TEX insertions.



- `parallelogram(P, U, V)`

A parallelogram is parametrized by a corner P together with two vectors U, V ranging out from it, as a box is determined by corner, width, and height. The path constructed is closed.

- `polygon(p)`

Constructs the polygon p, which is an array of points (P_i). It does not close it.

- `grid(N, ds)`

This constructs a grid $[-N, N] \times [-N, N]$ of squares $ds \times ds$. It is meant primarily as a simple debugging utility. If in the course of drawing something you want to display temporarily what your coordinate system looks like, you can include something like:

```
newpath()
grid(5, 1)
stroke(0.8)
```

It is easy to delete when you are through 'debugging' your picture.

- `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))
```

Note the use of float to prevent problems with integral division. 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**.

8. Graphics states I. The coordinate system

Your graphics environment or graphics state changes throughout your program. At any moment it records

(a) the current coordinate system, which keeps track of the relation between the programmer's coordinates and those of some fixed default coordinate system;
(b) the current color;
(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 temporarily, only to go back to the old one after a while. To allow this, both PostScript and πS maintain an array of graphics states which one manipulates from time to time. It is a stack, as they explained to you in your beginners' course in programming. 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. Coordinate changes thus accumulate as the stack expands.

The most important part of the graphics state records the transformation from current user coordinates to the default system.

- `center()`

This is the simplest coordinate change of all—it translates the origin of the coordinate system to the center of the bounding box. It is usually a good idea to do this at the beginning of every page.

- `gsave()`

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

- `grestore()`

Restores the previous graphics state.

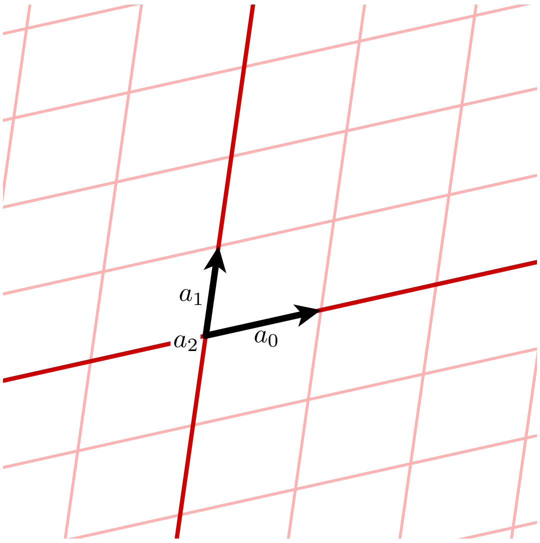
> *It is a very good idea to sprinkle* `gsave()/grestore()` *pairs liberally around in a program, encapsulating just about every graphics object you are dealing with.*

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 `gsave`s or `grestore`s. *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.

We have seen several commands with coordinates as arguments (such as `moveto`). These are the coordinates of points expressed in **user coordinates**, and in the process of drawing user coordinates are transformed immediately to default coordinates (and then in turn sooner or later 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 its x- and y-axes. A frame on the plane determines the coordinates of every point on it.

A frame is determined by a point and two vectors. By convention, the origin of the frame is labelled as a_2, the unit vector along the x-axis as a_0, and that along the y-axis as a_1. We'll see later the reasons for this odd indexing.



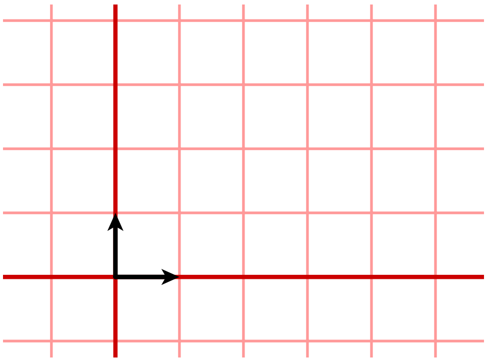
It is important to keep in mind that **coordinate changes act by changing the frame.** We'll see how this works in the examples below, and coordinate systems are also discussed later on in the section on affine transformations. It's so important that I'll repeat it:

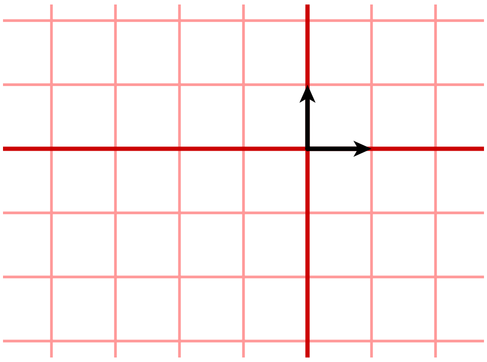
Coordinate changes move the coordinate frame.

Now for examples.

- `translate(V)`

Translates the origin of the coordinate system by the vector V. Thus `translate(3,2)` has this effect:

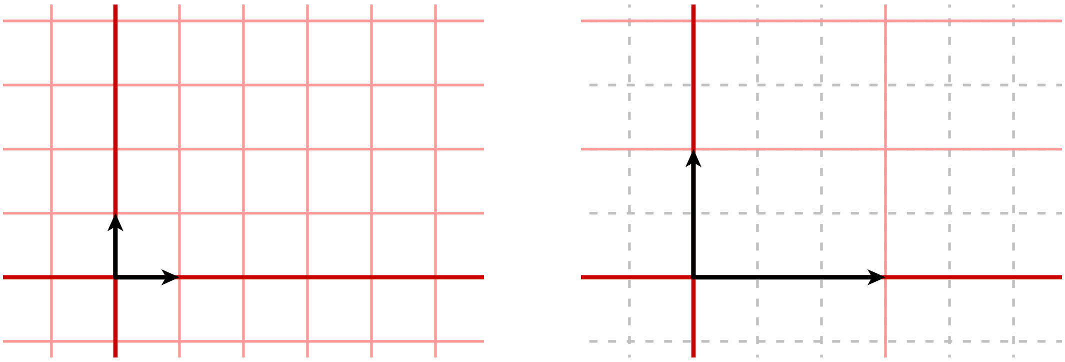




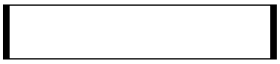
The point which used to be $(3, 2)$ is now the origin in the new coordinate system. The unit vectors along the axes are the same.

- `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. In PostScript, line width scales along with all other dimensions, and if the scale is different in x and y directions, line width will vary with the direction of the line. But in πS this is not true. *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*. 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 πS 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. Here is `scale(3,2)`:



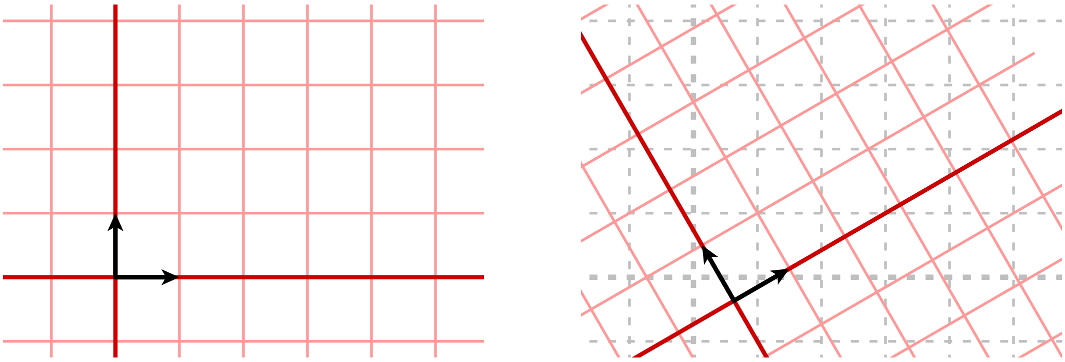
and here is what happens in PostScript when you draw a square after a non-uniform scale change:



Intriguing, but probably not your heart's desire for everyday fare.

- `rotate(a)`
  `rotate(P, a)`

Rotates the current coordinate frame by angle a around the point P (or around the origin if P is not specified). Here is the effect of `rotate(P, math.pi/6)` where $P = (1, 1)$:

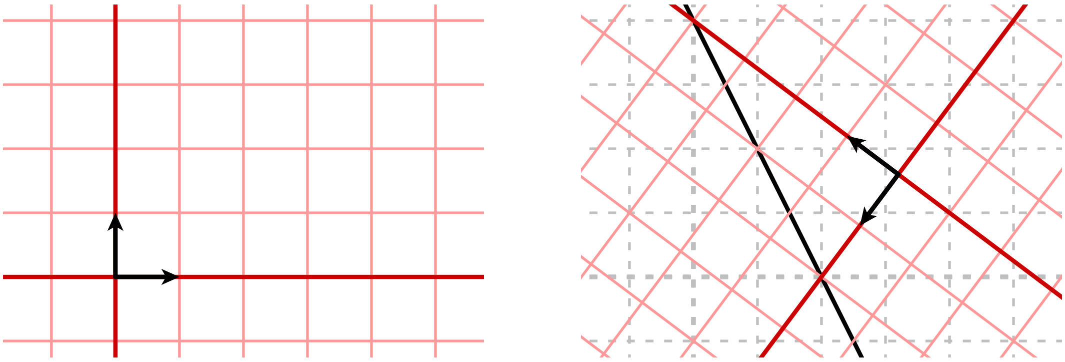


- `reflect(f)`
  `reflect(f, v)`

Reflects the current coordinate frame in the affine line $ax + by + c$, where $f = \{a, b, c\}$. If v is not specified, it assumes v to be perpendicular to $f = 0$ with respect to the Euclidean metric. That is to say if $f(a, b, c)$ then $v = [a, b]$. In any case, reflection takes v to $-v$. This can be used in conjunction with the method

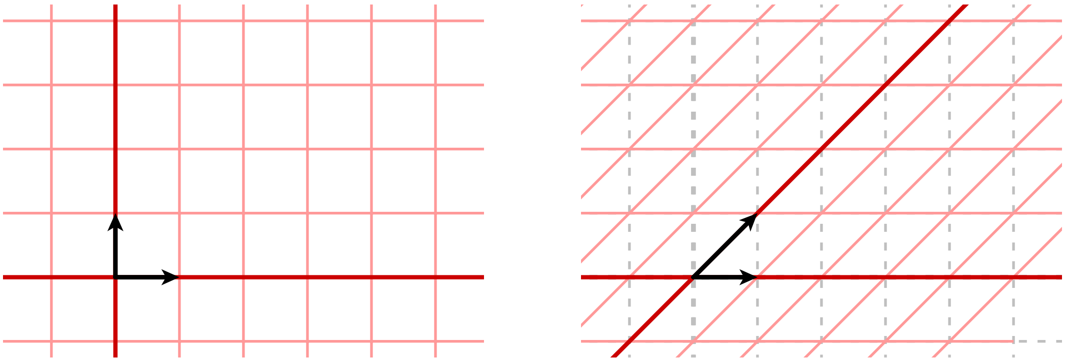
- `linethrough(P, Q)`

This returns (a, b, c), where $ax + by + c = 0$ is the equation of the line through P and Q. The vector $[a, b]$ is of unit length. Here is the effect of `reflect([2,-1,4])`, which reflects in the line $y = -2x + 4$:



- `atransform(a0, a1)`
  `atransform(a0, a1, a2)`
  `atransform(a)`

Applies an affine transform specified 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 translation 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 argument is a single array, it specifies either $(a_{0,0}, a_{0,1}, a_{1,0}, a_{1,1})$ or $(a_{0,0}, a_{0,1}, a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1})$ A shear along the x-axis, for example, is `atransform([1,0],[1,1])`:



Coordinate systems in PostScript and πS 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 of that book. I'll say more about this later on.

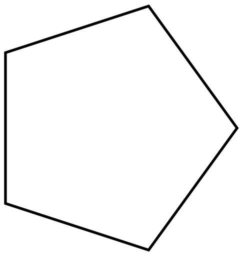
I summarize: the coordinate change operations are `scale`, `translate`, `rotate`, `reflect`, and `atransform`. Actually, there is a mathematical theorem that asserts that all we absolutely need are translations, scalings, and linear rotations, but that would be awkward to depend on. It ought to give the idea, however, that you should be careful when combining coordinate changes, because you can in principle 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/2)
lineto(1,0)
```

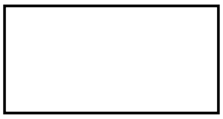
produces a line between the points that were $(1,0)$ and $(0,1)$ in the starting coordinate system. And thus

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



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

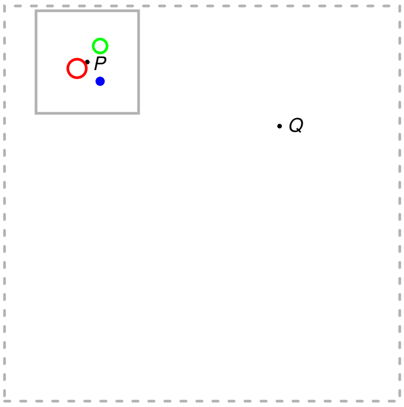
```
scale(2, 1)
newpath()
box(1,1)
stroke()
```

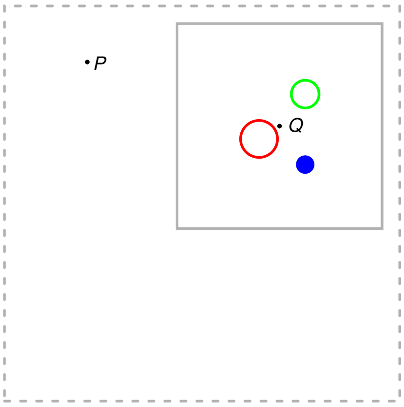


Sometimes after drawing a complicated figure, you realize it's not placed quite correctly. Maybe what you want to do is zoom into your figure to emphasize some particular feature.

- `zoom(P, Q, s)`

This changes coordinates so as to zoom in on a part of your figure (or to move out from it). The scale change is s, and what is currently P is moved to what is currently Q.

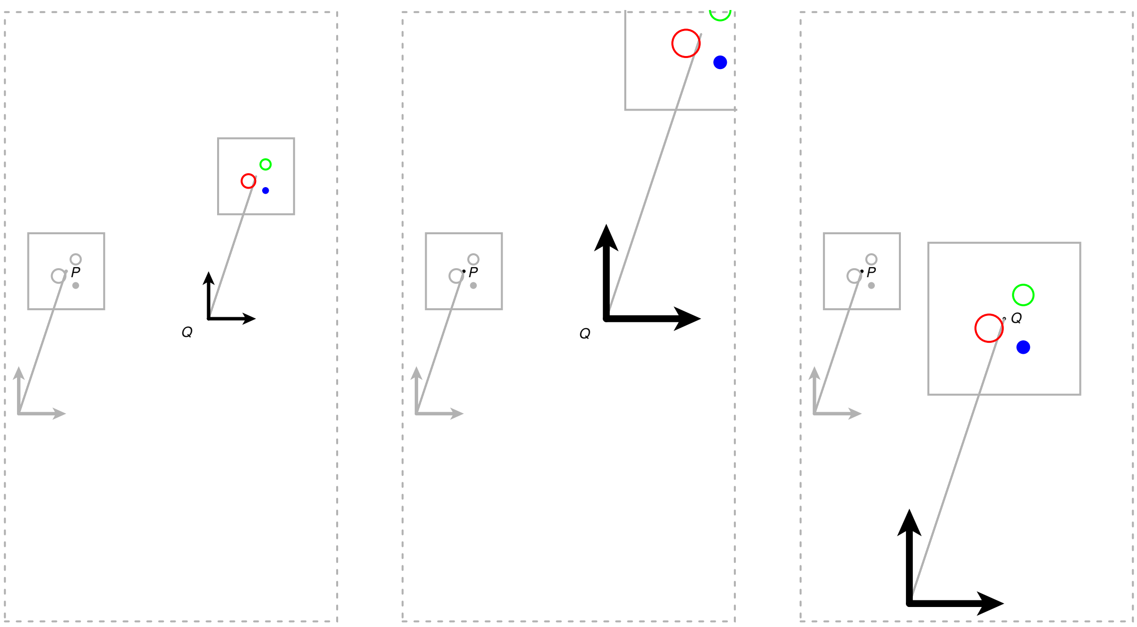




This is a simple combination of elementary coordinate changes:

```
def zoom(P, Q, s):
    translate(Q)
    scale(s)
    translate(-P[0],-P[1])
```

explained by these pictures:



Often you will want to move something to the center of your figure. For this you can use

- `currentcenter()`

This returns the center of your figure in current coordinates.

Sometimes you will want to revert to the default coordinate system. This is done in slightly different ways by the commands

- `revert()`
- `lrevert()`

The first reverts back to the original default coordinate system, with lower left corner as origin, basic unit 1 Adobe point. The second keeps the current coordinate origin, but sets the linear coordinates to the default. Usually, as we'll see in examples, you'll want to revert only temporarily to default coordinates, so you should encapsulate reversions in `gsave()/grestore()`.

Both of theme return a copy of the **graphics state** current at the time it is called. Often you will want to use this while you are in default coordinates, because it gives you access to the geometry of the coordinate system you were using when you reverted. The basic idea is that

```
gs = revert()
P = gs.transform(1,1)
```

sets P equal to (x, y), where x, y are the coordinates in the default coordinate system of the point that was $(1, 1)$ when `revert` was called. For example, `lrevert` is equivalent to

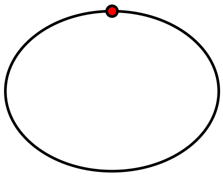
```
gs = revert()
translate(gs.transform(0,0))
```

The way this works is that 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 that control how drawing is done, such as the current line width. More precisely, either reversion returns an instance of a class `GraphicsState` that specifies these. Probably the only thing you have to know about it is that

- `(GraphicsState).transform(P)`

returns the transform of P into default coordinates by the coordinate transformation of the graphics state.

Why would you want to use reversion? Suppose you want to draw



The ellipse is easy to draw—you scale non-uniformly in x and y, and then draw a circle. But now you want to place a small disk on top of the ellipse. So, let's see: you want to draw a circle at the point whose coordinates are $(0, 1)$ in the coordinate system you have set up. But you want it to be a true circle, whereas if you draw a small circle at $(0, 1)$ without doing something tricky you'll get another ellipse. This is the cue for reversion:

```
scale(40, 30)

newpath()
circle(1)
stroke()

gsave()
translate(0, 1)
lrevert()
newpath()
circle(2)
fill(1,0,0)
stroke()
grestore()
```

This raises a rather philosophical point. In mathematics, some parts of your figure are really part of the geometry of the figure, like the ellipse in the figure above. But some parts are what I call **meta-graphics**—clues as to how to interpret the figure. The disk at $(0, 1)$ is probably one of these—the disk is something in the figure that marks a certain place in it. It doesn't have to be a disk, it just has to be small but visible. After all, the point it is marking has, as Euclid tells us, no dimensions at all. So the disk is a kind of label, with symbolic but not literal meaning in the figure, like the vertex labels A, B, etc. in geometry texts. The primary example of meta-graphics is text in the figure.

9. Interlude—Pythagoras revisited

With coordinate changes and the utility `box`, drawing the Pythagoras configuration is a bit more elegant:

```
newpath()
box(-a, a)
stroke()

newpath()
box(b, -b)
stroke()

gsave()
translate(0,a)
A = math.atan2(a, b)
rotate(-A)
```

```
box(c, c)
stroke()
grestore()
```

Why do I think it's elegant? Good programming is flexible and easy to understand, therefore easy to modify for reuse. I think this fragment qualifies, even though it is elementary. The previous short code was good only for squares, whereas here we are using `box`. And the use of a coordinate change to draw the square at an angle is a trick I find generally useful, and in this case as in many others intuitive in the sense that it follows closely how we perceive the figure.

10. Graphics states II. Other features

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

- `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 the range $[0, 1]$. Gray g is equivalent to (g, g, g). THe current color is used in commands that require color but do not set it themselves. These include `stroke()`,`fill()`, and text display.

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

I should mention that the clipping path is also part of the graphics state.

- `scalelinewidth(c)`

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

- `setlinewidth(c)`

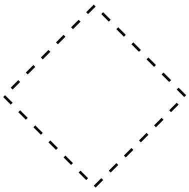
Sets the line width. *The unit in which line width is measured is always points.* It is equal to 1 point unless changed with one of these two commands.

- `currentlinewidth()`

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

- `setdash(a, b)`

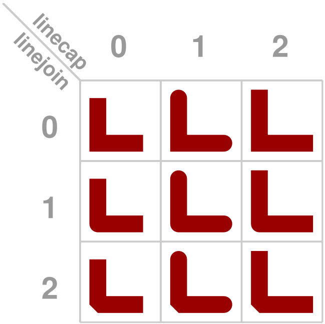
Sets current stroke pattern to a dashed line. Here a is an array of lengths setting the on/off pattern, b is the initial offset. In the following figure, the side of the square is 1, $a = [4, 4]$, $b = 0$.



The dimensions of the arguments are points, since stroking is rendered in default coordinates.

- `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 (this is done by default in the πS 3D package) and sometimes you want $n = 1$ for drawing arrows, but otherwise the default 0 is best. Here are the effects:

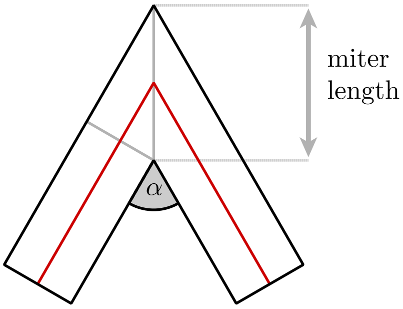


The style with `linejoin` set to 0 is called a **miter** style, that with 1 is **round**, and that with 2 is **beveled**.

- `setmiterlimit(x)`

This is the most technical of all of these commands. It 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.

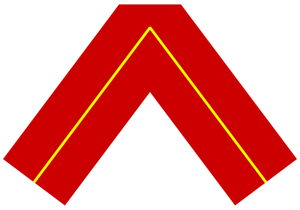
Suppose two lines join at angle α. The following diagram defines the **miter length**.

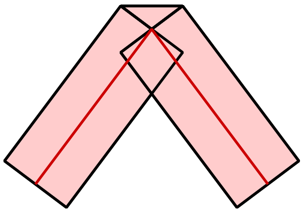


The diagram also shows that

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

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

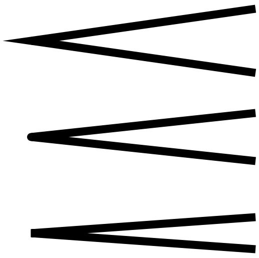
$$\frac{1}{\sin(\alpha/2)} = x, \quad \alpha = 2\arcsin(1/x)\,.$$

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).



11. Arrows

Arrows are an extremely, maybe surprisingly, common feature of mathematical diagrams. I have therefore tried to make the package flexible and easy to extend.

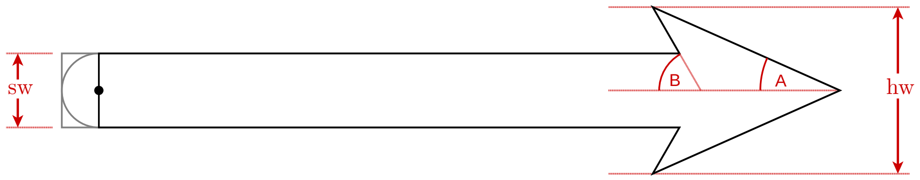
There is a philosophical problem with arrows. Are they geometry or meta-graphics? On the one hand, nearly every use of an arrow will be symbolic, but on the other they are almost always tied closely to the geometry of the figure they are embedded in. So there is no completely satisfactory answer. By default, they are geometry—scaling dimensions will scale arrows in every way. How could it be otherwise? How could you scale an arrow's length without scaling its width? Reversion is one way to deal with this.

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 $A = 24^\circ$, $B = 60^\circ$. 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`.

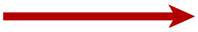
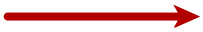
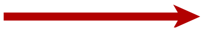
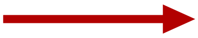
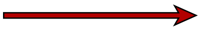
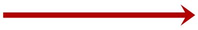


All arrows go from $(0, 0)$ to P, so normally you'll apply coordinate changes when using them.

- `arrow(V)`
- `openarrow(V)`

The first 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.

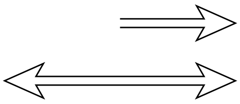
Here are some samples, with various values of sw and hw. In the last one, $A = 48°$, $B = 60°$.

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()
```



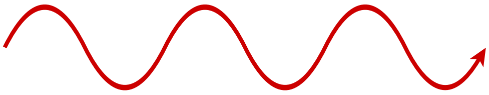
- `quadarrow(a)`

Builds an arrow using the argument a to assemble quadratic Bézier paths. The argument a is an array 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)
```

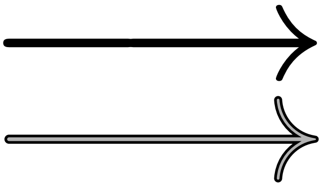
(in suitable units) produces



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(P)`

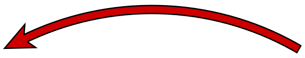
This creates an arrow that looks like those produced in TEX, commonly used in commutative diagrams. The shaft width is set by `setarrowdims`. In the figure below, the arrow on top is a magnified `\longrightarrow` from TEX, the one on the bottom produced by `texarrow`.



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!

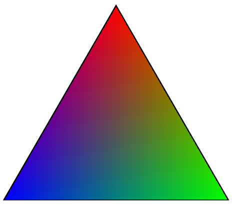
- `arcarrow(P, Q, r)`
  `arcarrow(P, r, A, B)`
- `arcnarrow(P, Q, r)`
  `arcnarrow(P, r, A, B)`

The first forms build 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. The second ones build it as an arc of a circle centered at P.



12. Gradient fills

You can obtain curious effects such as this:



with

- `shfill(data)`

Here `data` is an array of 'colored triangles'. Each colored triangle is itself an array of 'colored vertices'. A colored vertex is an array `(P, C)` where $P = (x, y)$ and $C = (r, g, b)$. Each triangle is filled in by a spread of colors interpolating those at the vertices. We shall see a good application of this in 3D drawing, in order to give an illusion of a smooth surface, but it is occasionally useful even in 2D. Almost always, the (colored) vertices are built first, then the triangles assembled from them, so colors of neighbouring triangles are consistent.

This is the simplest kind of gradient fill implemented in PostScript, and the technique applied in 3D is called **Gouraud** shading. The significant part of the code that produced the figure above is

```
P = [ ( 0, 1), (1, 0, 0) ]
Q = [ ( c,-s), (0, 1, 0) ]
R = [ (-c,-s), (0, 0, 1) ]
ds = [[P, Q, R]]
shfill(ds)
```

where c and s are cosine and sine of $30°$.

13. Under the hood

There are a huge number of commands to use in πS, but out of sight some kind of distillation is taking place. In the end, a πS program boils down to a sequence of relatively simple commands with simple parameters. I have already told you neither more nor less exactly what πS does: (a) it constructs paths and then (b) it either strokes them, fills them, or incorporates them in the clipping path. Not in fact all that complicated, if you think about it. So what I am going to tell you here should not be too surprising.

The major function of a πS program is to produce a list of commands and parameters. The commands make up a very small list, and can be grouped according to task.

Constructs paths
- `newpath`
- `moveto`
- `lineto`
- `curveto`
- `closepath`

Strokes, fills, clips
- `fill`
- `stroke`
- `cfill`
- `cstroke`

```
clip
shfill
```

Manages the special paths that are text

```
setfont
show
```

Controls how stroking etc. take place

```
setlinewidth
setlinecap
setlinejoin
setdash
setmiterlimit
setcolor
scalelinewidth
gsave
grestore
```

Miscellaneous

```
embed
insert
importps
importeps
comment
```

We have seen something of most of these. It's not a very long list, and in fact it is even a bit redundant. There don't have to be two kinds of stroking and filling, one that uses the current color and one that sets its own, but it seemed to me slightly more efficient to have them.

The virtue of this simple scheme is that, at least in principle, πS can output to a wide range of devices, including for example `PDF`, not just PostScript.

One thing to notice is that there are no coordinate changes among the commands. As your program churns along, all coordinates are rendered immediately in the default coordinate system. So, for example, an initial sequence

```
scale(2)
translate(1,1)
newpath()
box(10, 10)
stroke()
```

becomes

```
newpath
moveto 2, 2
lineto 22, 2
lineto 22, 22
lineto 2, 22
closepath
stroke
```

Sure enough, all coordinates in the final array are in the initial default coordinates. What is also true, although not evident in this case, is that the output is verbose. For example, the single command `beginpage` expands to

```
gsave
setlinewidth 1
```

```
setlinecap 0
setlinejoin 0
setdash [], 0
setmiterlimit 10
setcolor 0, 0, 0
newpath
moveto 0, 0
lineto 100, 0
lineto 100, 100
lineto 0, 100
closepath
clip
```

Why am I telling you this? Well, sometimes, if very rarely, you can figure out what's wrong with your program by scanning this low-level translation. You can get it by defining a variable to be the return value from `init`: `ps = init(...)` and then when you feel like seeing what you are getting `print ps`. This is how I produced the examples above! Command arrays as I describe here are really at the core of πS. THe return value from `init` is an instance of the Python class Canvas, as are the return values from other commands. The main function of a Canvas is to hold a command array. Eventually, πS will allow you to draw Canvases in a wide variety of configurations.

Part 2. Text in figures I. TeX

There are two quite different ways to put text in πS figures. One allows you to embed TeX in 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. These two different approaches are being slowly merged in successive versions of πS.

14. Placing TeX Text

The basic object for placing TeX labels into a figure is a `TexInsert`. It has an existence independent of where it is to be placed, and includes in its data dimensions (such as width) and its own coordinate system, as well as the text to be placed.

- `texinsert(texstring)`
  `texinsert(texstring, label)`

These create and returns a TexInsert object. The optional label is a string that will function as part of the name of the TeX and `.dvi` files produced. Often you will use this command to assign a variable (as in `t = texinsert("$x$")`), and then do something with it.

There is one extremely important thing to watch out for when making TeX inserts:

In writing TeX macros in a string that's 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` in a Python string means 'linefeed', and is used in Linux systems to mark the end of a line of text. For example, 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.

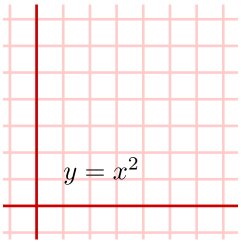
Usually an inserted fragment of TeX is to be considered meta-graphics—it has symbolic meaning, and is not part of the true graphics—but sometimes not. Depending on how it is to be considered, insertion takes place with one of the two commands `place` (symbol) or `embed` (geometry).

- `place(t)`
  `place(t, P)`

Places the TexInsert t in your figure, considering it as meta-graphics. A TexInsert has an origin, which is by default at the origin of the first character in the TeX string. With the second option πS puts this origin at the point P (specified in current coordinates). If you use the first option, not specifying P, it is placed at the current coordinate system origin.

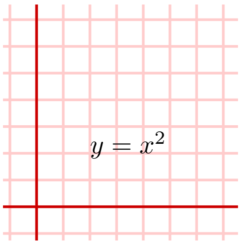
The simplest usage:

```
place(texinsert("$y=x^{2}$"), 1, 1)
```



(Here and in following figures there has been a preliminary `scale(10)`, so that each square in the unit grid is 10×10 points.) The command `place` places a TexInsert *scaled according to default coordinates*. (As we'll see in a moment, this is another constructive lie.) Thus in the following case, the appearance of the TEX insert is the same as it was before, except that scaling user coordinates moved its origin:

```
scale(2)
t = texinsert("$y=x^{2}$")
place(t, 1, 1)
```

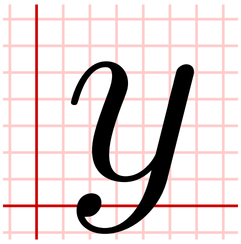


We'll look at more examples in a moment.

- `embed(t)`

This embeds `t` as a graphics object, applying current coordinate changes to it. You have to be pretty careful with it. In the following figure, there was an initial `scale(10)` (off screen, so to speak) to make the grid of 1×1 squares. Now TEX fonts are themselves used at a nominal 10 point size, and with a scale factor of 10 this makes a font of nominal sioze 100 points. So the TEX at 10 points comes in rather large. Note that there is no `embed(t, P)` option.

```
t = texinsert("$y=x^{2}$")
translate(1, 1)
embed(t)
```

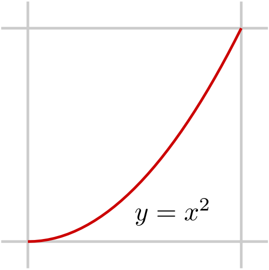


The point is that text in figures is not normally part of the graphics, but is what I call part of its *meta-graphics*. It is for this reason that the default behaviour of πS, with how it shows TEX inserts as with how it strokes lines, is to place it in the default coordinates. We shall see, however, that even when TexInserts are symbolic there is a use for `embed`.

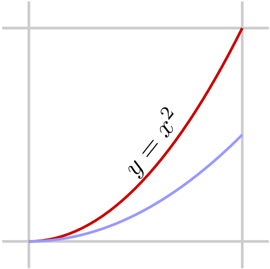
15. A TEX insert as a graphics object

Now to tell you how I lied.

Let me first explain the problem that we have to deal with. Suppose you have graphed the function $y = x^2$, and you want to label the graph. You could just do something like this:



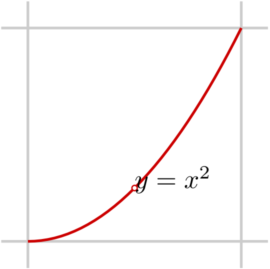
but then if you decided also to add the graph of $y = x^2/2$ you'd have to move the label out of the way of the new graph. And if you wanted also to label the new graph, you'd probably think about this for the original label:



This in fact illustrates a good general principle—*labels should be placed near what they are labeling.*

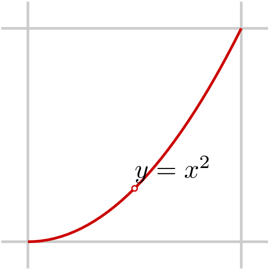
But how can we do this? We'll see later one solution, but in this situation it is overkill. Here there is a relatively simple solution, but one that uses a new trick. *We apply geometric transformations to the TexInsert!* I explain in several steps. First we place out text at some point on the curve.

```
t = texinsert("$y = x^2$")
place(t, 0.5, 0.25)
```



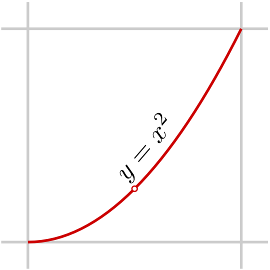
But now before doing that, we translate the TexInsert up by 4 points.

```
t = texinsert("$y = x^2$")
t.translate(0, 4)
place(t, 0.5, 0.25)
```



And finally, after we do that translation we rotate the TexInsert.

```
t = texinsert("$y = x^2$")
t.translate(0, 4)
t.rotate(0.86)
place(t, 0.5, 0.25)
```



In effect, each transformation changes the TexInsert, but **not** its origin. Think of it as pushing the contents of the Insert around inside a fixed region.

You can change the size of a TEX insert, too, and change its shape in other ways. Here are the allowed transformations:

- `(TexInsert).translate(...)`
- `(TexInsert).scale(...)`
- `(TexInsert).rotate(...)`
- `(TexInsert).reflect(...)`
- `(TexInsert).atransform(...)`

The dots `...` mean that the arguments are the same as they were for earlier coordinate changes. However, *the meaning of these is very different from the coordinate changes.* Here, we are transforming the contents of the TexInsert around, whereas before we were changing the current coordinate system in which we were drawing. I'll explain in more detail later on, when I discuss coordinate systems, how this works.

16. Saving for the future

When creating a TeX insert, ΠS (1) creates a TeX file, (2) runs TeX on it, and then (3) processes the `.dvi` file this produces to get a command array, which (4) it puts in a TexInsert object. By default it will then delete the TeX and `.dvi` files, which it doesn't need any more. You can direct ΠS to save these, however.

- `settexsave(b)`

The argument here is either `True` or `False`, and while saving is set to `True` then TeX files produced by `texinsert` will be saved. The advantage of this is that when the same program is run again, it will compare the TeX file it produced on the last run with the one it would create on this one, and if the two are the same it won't run TeX again. Instead of producing a new `.dvi` file, it just reuse the `.dvi` file it sees. Running TeX is one of the more expensive parts of ΠS (in currency of time), and this can make your program run faster.

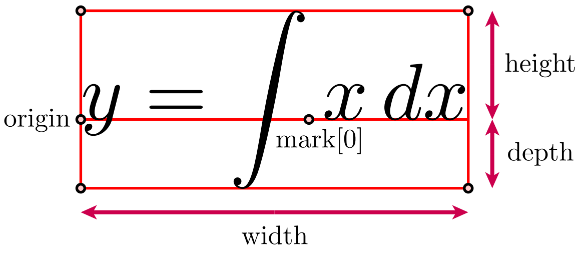
File saving takes place in one of two ways. One is to set a global variable `texsave` to be `True`, in which case every `texinsert` will save stuff. Or, you can use the optional label in `texinsert`. In the first case the saved files will be assigned file names starting with `tmp` and incorporating successive numbers, except that the labeling option always overrides this. Of course, you can revert to not saving files with `settexsave(False)`.

If files are saved, *it is up to the user to get rid of obsolete TeX and* `.dvi` *files later on.* Not running TeX unnecessarily speeds up things noticeably, although comparison of old and new TeX files is not extremely fast. In TeX inserts ΠS will usually 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.

17. The geometry of a TeX insert

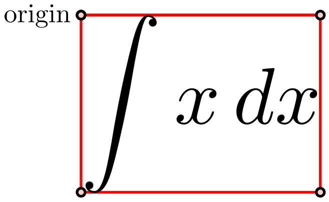
A TeX insert also has several data attached to it in addition to its text, mainly specifying certain points of its geometry.

(a) `bbox`
(b) `width`
(c) `height`
(d) `depth`
(e) `origin`
(f) a set of locations stored in an array `mark`

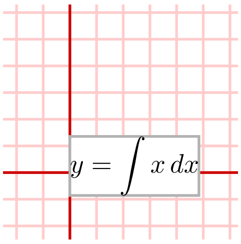


I'll explain the mark later, but I should tell you right now that the TeX placed here is `$$y = \int \mark x \, dx$$` A TeX insert has its own coordinate system, in which the origin is by default the left bounding point of the first character that's put into it. This location of the origin works well to allow alignment of different texts.

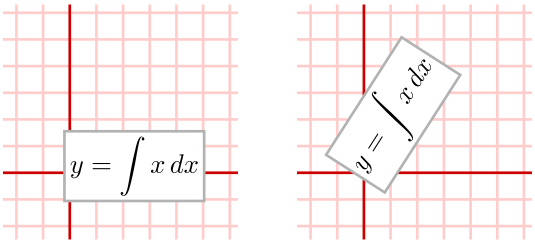
You have to be aware, though, that the origin of a character, in this case the integral sign, might not be what you expect:



Since this is probably not where you want the origin to be, here is a good place where you want `\mark x` to help you locate your preferred origin on the baseline just under $x\,dx$. The coordinate system of a TexInsert is rectangular, and its unit is one point, at least initially. The height is the difference in height between the top of the bounding box and the origin, the depth is the difference in height between the origin and the bottom. All these data, including the bounding box `bbox`, are expressed in this local coordinate system. The bounding box can be used, among other things, to blank out space behind a TeX insert, as in this figure:

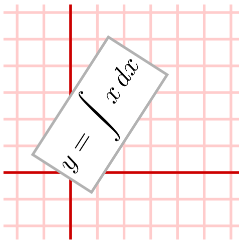


Normally, however, you want to blank out a bit more than just the tight bounding box, as on the left below. Or you might even want to subsequently move it around, as on the right:

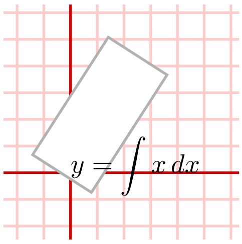


The good news is that you can do this, and in fact draw all kinds of things based on the 'internal geometry' of a TeX insert. The bad (well, not so bad) news is, doing so is not quite straightforward. There are several ways to do it, but I'll explain only one. As you might have guessed already, it's a good task for reversion to default coordinates. Here is how to produce the figure on the right:

```
t = texinsert("$$y=\int x\, dx$$")
gsave()
lrevert()
rotate(1)
b = t.bbox
ds = 2

newpath()
boundedbox(b[0]-ds, b[1]-ds, b[2]+ds, b[3]+ds)
closepath()
fill(1)
stroke(0.7)

embed(t)
grestore()
```



One mystery might be why `embed` rather than `place`. But `place` always places in the default coordinates, unless the TexInsert has been transformed. So with `place` instead of `embed` you get something you probably don't want:



18. The TeX environment

The way `texinsert` executes depends on the TeX environment in πS , which consists of three items—the TeX command (the name of a variant of TeX itself), the TeX prefix, and the TeX postfix. In this situation, a temporary TeX file is assembled from the prefix, your TeX string, and the postfix, and then the command is run on that file. These three items are normally read from a configuration file, but they can also be set dynamically.

The basic πS package contains the configuration file `TexConfig.py` in which a standard configuration is set, and this is the default. In this file the TeX command is set to “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 the postfix is

```
\end{document}
```

This means that when, for example, you put `t = texinsert("$y = x^2$)` in your program 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
```

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

You might be puzzled by the new command definitions, but I'll explain those later. The rest of this default prefix is pretty simple, a kind of basic minimum.

> *If this looks OK to you, at least for the moment, you can stop reading this section right now. Come back to it only when you are dissatisfied with your TeX environment in πS.*

There are roughly two ways to change your TeX configuration. One is to set the TeX prefix, command, or postfix in your program. The other is to change the configuration file that is read.

Here is a brief outline of what πS does to set up your TeX environment. (1) First of all, if you have set the prefix, etc. in your program it will use what you have set. (2) Otherwise, it looks for a configuration file with a certain name. This can be either the default name `TexConfig.py`, or it can be one you have specified. (3) Whatever the name of the configuration file that has been set, πS looks for a file of that name in one of three directories, in order: (a) your current working directory; (b) your own personal πS configuration directory; (c) the global πS directory.

I'll postpone explaining how to set the prefix etc. in your program, because using configuration files is usually much more convenient.

- Here is the code from the default file:

```
from piscript.Tex import TexEnv

def getTexEnv():
    p = "\\documentclass[12pt]{article}\n"
    p += "\\pagestyle{empty}\n"
    p += "\\input amssym.def\n"
    p += "\\newcommand\\color[1]{\\special{Color{#1}}}\n"
    p += "\\newcommand\\uncolor{\\special{unColor}}\n"
    p += "\\newcommand\\lmark{\\special{mark}}\n"
    p += "\\begin{document}\n"
    q = "\n%\n\\end{document}\n"
    c = "latex"
    return TexEnv(p, q, c)
```

A TeX configuration file must define a procedure `getTexEnv()` that returns the three strings that make up the prefix, postfix, and command. (Note here the use of `\\` for TeX macros.) My advice on how to make one up is to run TeX (by hand, so to speak) on several sample TeX files with different prefixes until you get one that you are happy with.

- When you first set up πS, you should make a directory to be your own 'local' πS directory, in which you put code that lots of your programs require. You should define an environment variable `LOCALPISCRIPTDIR` to be that directory. Inside that you should have a subdirectory `configs`, which in turn contains an empty file `__init__.py`. You should put your frequently used TeX configuration files in there. For example, on my Linux machine this is the hidden directory `.piscript` in my home directory, and the directory listing of `~/.piscript/configs` is

```
__init__.py
BoldMathTexConfig.py
PlainTexConfig.py
```

```
TexConfig.py
PSTexConfig.py
```

- The file `PlainTexConfig.py` mentioned above defines a Plain TEX environment with prefix, postfix, and command

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

\bye

tex 1> /dev/null
```

The command suppresses some (but not all) superfluous TEX output.

If you suppress all TEX output, then when there are errors in your TEX file they will manifest themselves by mysterious halts. In this case, pressing `Enter` *a few times will usually get things to come to a conclusion.*

Here, for example, is a file `PSTexConfig.py` that I use, calling for it with `settexenv("PSTexConfig")`.

```
def getTexEnv():
    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"
    p += "\\input psfont.defs\n"

    q = "\n%\n\\bye\n"
    # c = "tex 1> /dev/null"
    return TexEnv(p, q, c)
```

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
```

But I am not going to discuss in detail here how to use non-standard fonts in TEX. It is a potentially endless topic. All you need to know is that you ought to be able to reproduce your normal TEX environment exactly inside πS.

- You can do this with the commands

- `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 `LOCALPISCRIPTDIR/configs`, or the principal πS directory. These are searched in that order. The argument "plain" is equivalent to "PlainTexConfig", "latex" to "LaTexConfig".

In summary, you really do have just about complete control over how TeX is executed in πS.

19. Specials in 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}
```

In plain TeX, `mark` replaces `lmark`. A TeX file is created from a TeX insertion, and a TeX command, creating a `.dvi` file which is read into a program that creates an `.eps` file from it.

All this has been said before. What I want to say that's new is that certain TeX specials are allowed here. Right from the very beginning, Don Knuth realized people might want to expand TeX to give it extra capabilities. You have probably already seen examples, because embedding a PostScript figure into a TeX file is a special command. Other frequently used ones are involved in color changes, say with the package `colordvi`.

TeX specials are ignored by TeX itself and passed on to the next stage in interpretation. In particular, any TeX special you use in your program must not only be defined in TeX, but they must be interpreted down the line. What this means here is that specials you use in πS must be pre-defined in it. (Although now that I think of it, I foresee that sooner or later πS will allow you to write your own specials.) The ones currently interpreted are `mark`, `color`, and `uncolor`.

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 color macros in `colordvi.tex`, but I decided that the standard colors provided there weren't interesting enough, and that the cmyk (**C**yan, **M**agenta, **Y**ellow, blac**K** rather than **R**ed, **G**reen, **B**lue) conventions it follows weren't intuitive for most users.

Part 3. Text in figures II. PostScript

There are also available in πS a small number of methods that will place ordinary PostScript strings in your figure. In earlier versions of πS these methods were very closely tied to PostScript programming, and certain features were quite awkward. In the current version, they are tied to font handling in TeX.

These methods are not as versatile as using TeX to display text, but they are much more efficient, and often this is a more important consideration than fine control.

20. Placing PostScript strings

A basic description of what you can do with these methods is choose a current font at a specified size, then display simple text at locations you specify. What you do not have is sophisticated text placement.

- `setfont(f, s)`

Sets the current font to be f, with nominal size s. The size is usually interpreted in true points, as we'll see in a moment.There are a small number of fonts always available in PostScript, and in practice one should normally use only those. (Later versions of πS will make this advice obsolete.) They are

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

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

```
Courier
Courier-Oblique
Courier-Bold
Courier-Bold-Oblique
```

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
```

There are lots of other fonts available, too—all those that come with your TeX distribution. This includes all the basic TeX fonts such as `CMR10`, but probably also a large selection of less standard ones.

- `dimensions(s)`

Here `s` is a string. This returns the width, height, and depth of the string if placed in the current font at the current font size. It returns these as a triple `(w, h, d)`.

- `show(s)`
  `show(a)`

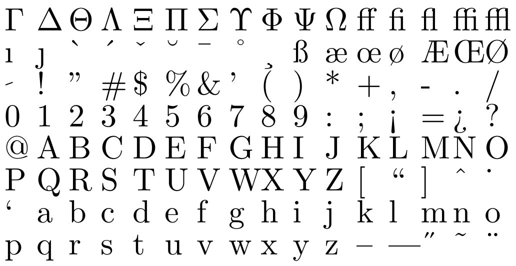
The first displays the string s at the current point in the current font. Normally this is immediately preceded by a `moveto`. The string is set under the same rules as TeX inserts to which `embed` is applied. The second is similar, except that `a` is an array of integers instead of characters. Thus

```
setfont("Helvetica-Bold", 12)
s = "Hello!"
d = dimensions(s)
moveto(100,100)
rmoveto(-d[0]/2, -d[1]/2)
show(s)
```

places the string “Hello” in Helvetica-Bold nominal size 12 points, centred at the point $(100, 100)$.

The following program will let you look at a whole segment of a font (in this case, `CMR10`):

```
from piscript.PiModule import
*

init("cmr", 200,100)

beginpage()
cw = 12
ch = 12
gsave()
translate(0.5*width(),8*ch)
translate(-cw*8,-4)
setfont("CMR10", 12)
for i in range(128):
    m = i/16
    n = i % 16
    moveto(n*cw,-m*ch)
    show([i])
grestore()
endpage()
finish()
```


Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω ff fi fl ffi ffl
ı ȷ ` ´ ˇ ˘ ¯ ˚ ¸ ß æ œ ø Æ Œ Ø
- ! ” # $ % & ’ ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; ¡ = ¿ ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [“] ˆ ˙
‘ a b c d e f g h i j k l m n o
p q r s t u v w x y z – — ” ˜ ¨

There is a list of fonts nominally available in the file FontList.txt in the directory `examples` that comes with πS. You can update it for your system by running `listfonts` in that directory. On my machine, there are more than 600.

Part 4. Paths

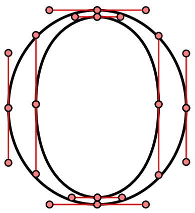
For some purposes, it is useful to build and store a path without drawing it.

21. Constructing and using paths

I have said that at bottom the output from every πS program is an array of relatively simple commands and parameters. There are a number of commands that take advantage of this fact, and go a bit further.

In the original version of πS, the program itself was tied closely to PostScript. This meant, among other things, that some of the output was definitely PostScript specific, in the sense that it relied on PostScript for certain tasks. One of these was a command that enabled the user to outline characters in a PostScript font, and along with it were commands to shift a displayed string in useful ways. These did not use πS structures for this, because πS did not have at that time practical capability to poke around in fonts. Much of the latest revision was to add this capability, at least to some extent. At the moment this part of πS is still changing, but the interface to it should remain the same through forthcoming changes.

Every character in a PostScript font is a path:

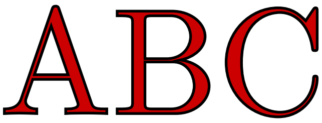


It can be accessed by the command

- `charpath(s)`

Here `s` is a string. This returns an array of the sort I have said πS puts out more generally. It amounts to an array of drawing commands that draws the outlines of the characters in the string. It is meant mainly as an internal method in πS that can be used to include the outlines of strings in any πS fragment, for example to assemble an entire TEX fragment as a single path. At the moment the interface to using it is . . . ugly.

```
def m(x, y):
    moveto(x, y)

def l(x, y):
    lineto(x, y)

def c(P1, P2, P3):
    curveto(P1, P2, P3)

def cl():
    closepath()

setfont("CMR10", 56)

newpath()
p = charpath("ABC")
p.set(MOVETO, m)
p.set(LINETO, l)
p.set(CURVETO, c)
p.set(CLOSEPATH, cl)
p.execute()
fill(0.8,0, 0)
stroke()
```



The ultimate point of this sort of thing is that one often one wants to draw some figure transformed, and it is straightforward 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 in πS are now paths of the sort I am talking about. For example, Arrows and arcs of circles are in this category. πS introduces 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. Strings are also in this category, and as I have said 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.

Another useful function is

- `dimensions(s)`

which returns an array (w, h, d), where w is the width of the string, h its height, and d its depth.

Part 5. 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.

22. 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.

Coordinate systems.

- `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.

Drawing 3D paths.

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

Starts a path.

- `rmoveto3d(V)`

- `lineto3dP`

- `curveto3d(P1,P2,P3)`

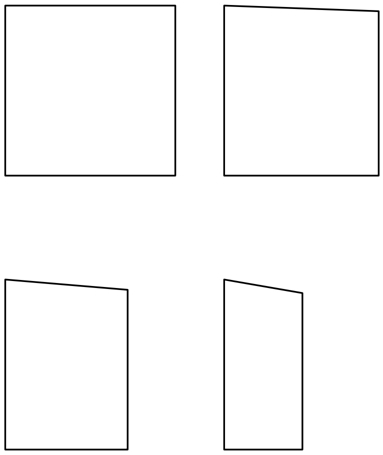
Here the arguments are 3D points, assumed to lie in a plane.

- `rlineto3d(V)`

- `closepath3d()`

Example. This shows on successive pages a rotating square frame, with the eye set at $(0, 0, 1)$:

```
init3d(".ps", 250, 150)

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

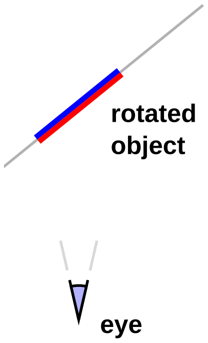


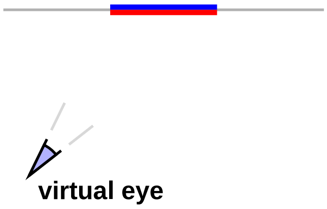
23. Visibility and lighting

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.ps", 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)`

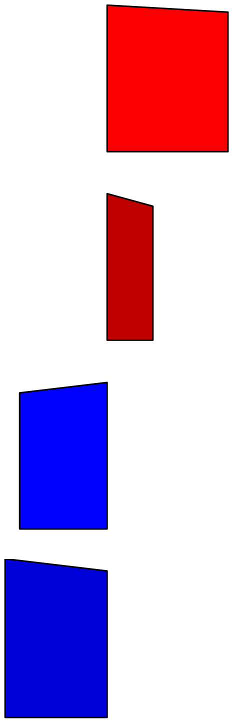
The vector L is also 4D, with last coordinate required to be 0. Sets the direction from which light comes. Internally, the light is a unit vector.

- `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.

```
init3d("litsquare.ps", 250, 150)

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()-
```



24. Convex surfaces

In πS 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. πS 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 πS) that the human eye is easily fooled, and in fact cooperates happily in being fooled.

πS 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 $[-1, 1]$—it is 1 if the light source is perpendicular & exterior 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. I'll say more about Bernstein polynomials later on in the section on shading. Here are the relevant functions:

- `convexsurface(f)`

Here f is an array of faces

- `paint(s)`

Here s is a convex 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)`

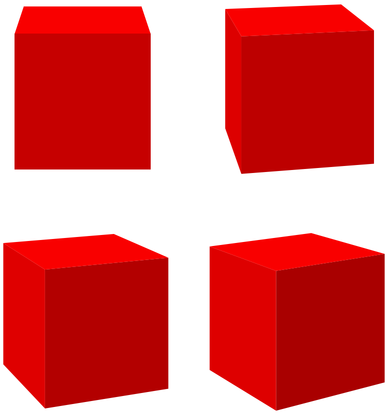
Changes the orientation of the face by multiplying its normal function by -1.

- `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 an example.

```
seteye([0,0,7,1])
translate3d(0,-2,0)

grey = 0
red = Cube([1,grey, grey], 1, [0,0,0])

for i in range(4):
    beginpage()
    center()
    scale(64)
    translate(0, 2)
    paint(red)
    endpage()
    rotate3d([0,1,0], math.pi/16)
```

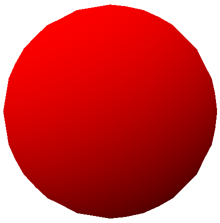


Put back in the rotating square.

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.

- `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. The parameter `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.

In the near future, πS will let you map any 2D picture into 3D. At the moment, it has limited but promising capability in this direction.

25. Shading

I'll now 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 $L = [a, b, c, 0]$, 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 what I shall call later an oriented homogeneous vector; equivalence of light sources is ruled by positive 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 determine 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 functions

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

The simplest function of this type is simply $d \mapsto (1+d)/2$. But this turns out not to be psychologically satisfcatory. What other good monotonic functions from the unit interval $[0, 1]$ to itself can one use?

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. Examples are the linear, quadratic, and cubic ones

$$\begin{aligned}&(1-t)\,y_0 + t\,y_1\\&(1-t)^2\,y_0 + 2(1-t)t\,y_1 + t^2\,y_2\\&(1-t)^3\,y_0 + 3(1-t)^2t\,y_1 + 3(1-t)t^2\,y_2 + t^3\,y_3\,.\end{aligned}$$

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 may be approximated by Bernstein polynomials, so that using them has a chance of not being a practical restriction.

These polynomials have a number of useful properties:

Theorem. *If y has length $n+1$ then*

(a) $B_y(0) = y_0$;

(b) $B_y(1) = y_n$;

(c) $B'_y(x) = nB_{\Delta y}(x)$,

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

The first two are elementary, and the last is only a mildly complicated calculation.

As a consequence:

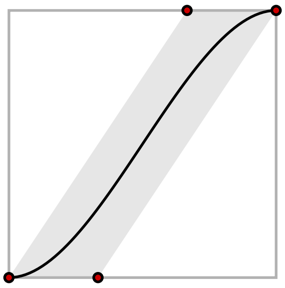
- $B'_y(0) = n(y_1 - y_0)$;
- $B'_y(1) = n(y_n - y_{n-1})$.

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

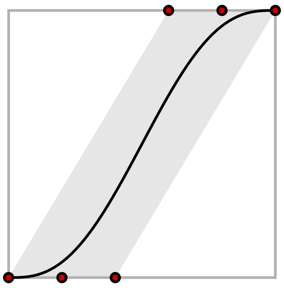
$$\big((1-x)+x\big)^n = \sum (1-x)^i x^{n-i} \binom{n}{i} = 1$$

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.

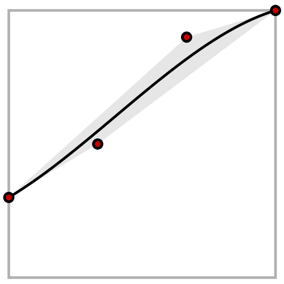
The figures below display the graphs of some Bernstein polynomials, with the y_i and the convex hull also indicated.



$\boldsymbol{y = [0, 0, 1, 1]}$



$\boldsymbol{y = [0, 0, 0, 1, 1, 1]}$



$\boldsymbol{y = [0.3, 0.5, 0.9, 1]}$

The default shading for convex surfaces in πS 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 πS that contains a small number of useful functions related to Bernstein polynomials.

Evaluation of Bernstein polynomials. Evaluating the polynomial $B_y(t)$ is not vastly different from evaluating an arbitrary polynomial value

$$P(t) = p_0 + p_1 t + \cdots + p_n t^n .$$

The preferred method is not to evaluate the powers t^k and then add, but to evaluate successively by Horner's method

$$p_n, \; p_n t + p_{n-1}, \; (p_n t + p_{n-1})t + p_{n-2}, \ldots$$

One modification required for Bernstein polynomials is that we deal with powers of both t and $1 - t$. Another is that we are given the y_i, not the coefficients of the polynomial, which we must compute on the fly. Recalling that

$$\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$$

we come up with:

```
def bernstein(y, t):
    n = len(y)-1
    t = float(t) # to eliminate Python's problem with integer division
    s = 1-t
    p = 0
    k = 0
    c = 1
    for a in y:
        p = p*s + c*a
        c = c*(n-k)*t/(k+1)
        k = k+1
    return(p)
```

The file `Bernstein.py` also contains functions

- `delta(y)`
- `sigma(y)`
- `subdivide(y)`
  `subdivide(y, n)`

In each of the first two, $y = (y_i)$ is an array of real numbers. If $n+1$ is the length of y, they return arrays of length $n-1$ and $n+1$:

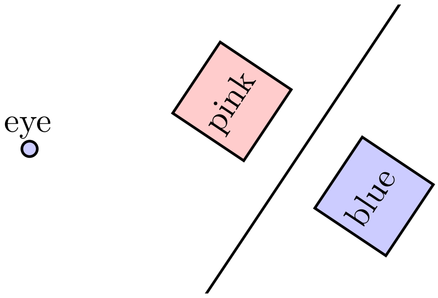
$$\begin{aligned}(\Delta y)_i &= y_{i+1} - y_i \\ (\Sigma y)_i &= y_0 + \cdots + y_{i-1} .\end{aligned}$$

In the last, y is an array of 4 2D points y_i. It therefore corresponds to a cubic Bézier curve. It returns an array of the 2^n arrays of control points obtained by recursively bisecting this curve into smaller Bézier cubic curves. This is useful when mapping paths by highly non-linear transformations. Here $n = 1$ if not specified.

26. How to deal with several objects

So far, what we have seen how to do in 3D is draw a single convex object. But drawing non-convex objects or even several convex objects is much more difficult, and in fact the techniques currently available in πS are not well adapted to this task. The problem is that one object can hide another, or at least part of another. What this amounts to is that the object that is hidden must be drawn first. The professional 3D programs accomplish this by maintaining 3D graphics data in a z-buffer. Without it, drawing really complicated 3D objects and dealing with effects like transparency are hard, even nearly impossible.

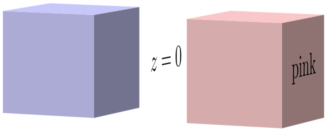
Still, something can be done. Let's start with a simple example, that of two cubes in space, originally each of side 1, originally centred at $(0, 0, \pm 1)$, but then rotated. Assume the eye to be out along the positive z-axis. We can tell very easily which is in back, which in front: the plane $z = 0$ separates the two cubes, as does this plane after it is rotated along with the cubes. The cube that's in back is the one on the side of the rotated $z = 0$ plane away from the eye. Equivalently, it is the one whose original position is on the side of $z = 0$ opposite to what I have called the virtual eye.

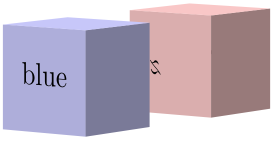


So the important code that draws the cubes in the proper order is

```
e = geteye() # e = the virtual eye
if e[2] > 0:
        # i.e.  if the virtual eye is on the positive side of the xy-plane
        paint(blue)
        paint(pink)
    else:
        paint(pink)
        paint(blue)
```

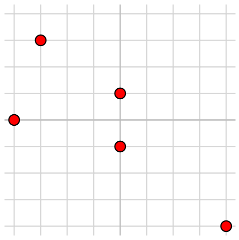
and as we rotate we see (after some decoration):



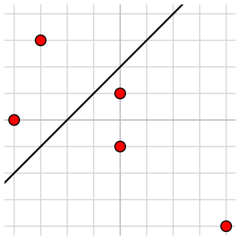


This is the simplest example of a **binary space partition**.

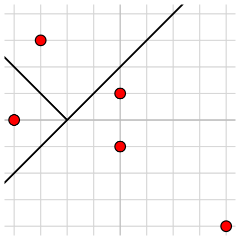
Let's look at a slightly more typical example. We start with a collection of a finite number of points at which we intend to place cubes:



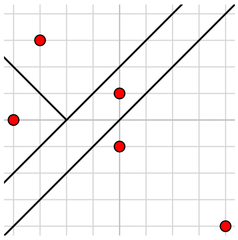
We then divide the points in two parts by means of a single straight line:

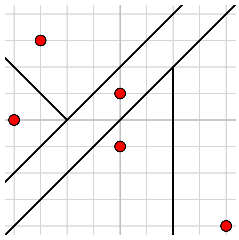


Next, we divide one of the two parts into two halves:

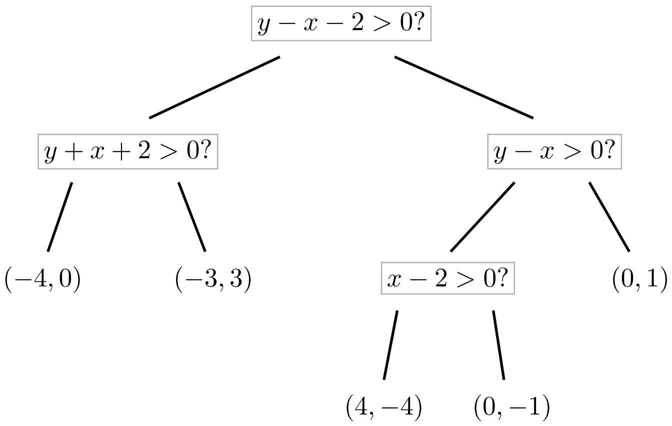


And so on:



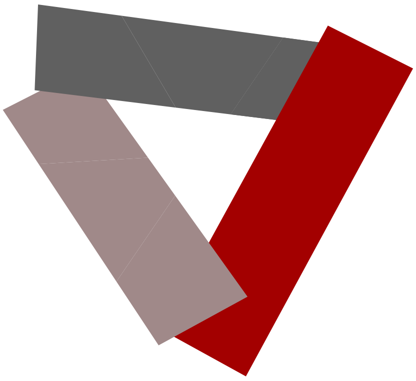


The structure implicit here is a tree:



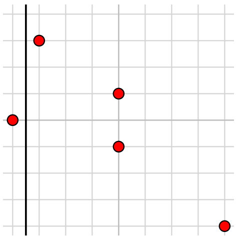
The line $y - x - 2 = 0$ divides the plane into two halves, and then successive equations divide those halves, and so on, except that when just one point lies in a region we stop. In order to draw the assembly we get, we traverse the tree, checking at each equation node which half to draw first by checking whether the equation of a node is positive or not on the virtual eye. We get in this way a partition of the plane, but in 3D all of space, and each node is binary.

This idea works easily if we are given a collection of points and want to partition space into convex regions with each one containing one point from the collection, but for drawing more complicated collections of objects it fails. In some cases, as in the following example, there is no strictly back-to-front way to draw things, and one must chop up the objects first. I'll say no more about that here.

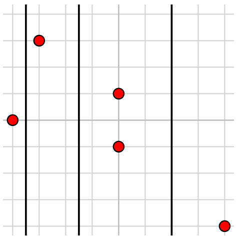


The technique suggested for a collection of points is limited, but it does allow some interesting graphical constructions. The most interesting of all is probably that of graphs of function $z = f(x, y)$. In the obvious version, this is restricted to the graph surface over rectangles, but in many cases other regions can be considered as well.

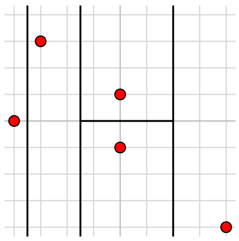
First I'll make somewhat more explicit a method for partitioning a real vector space in the presence of a finite collection of points. We first sort the points lexicographically: $x = (x_i) < y = (y_i)$ if the first coordinate in which x and y differ is less for x than it is for y. Thus $x = (1, -1, 1) < y = (1, -1, 2)$ because $x_1 = y_1$, $x_2 = y_2$, but $x_3 < y_3$. We then partition space according to the first coordinates. In the example from the images shown previously we first divide the plane into the regions $x_1 + 3.5 < 0, x + 3.5 > 0$.



We continue in this way, breaking off successively regions where points all have the same value of x_1:

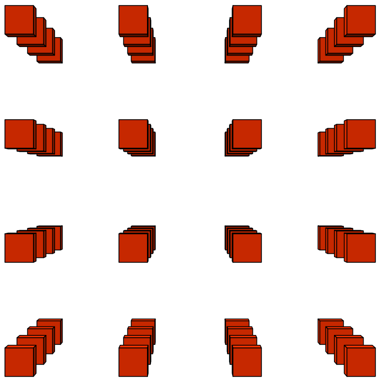


If a piece has more than one point in it, we partition it further, according to the sorted values of x_2:

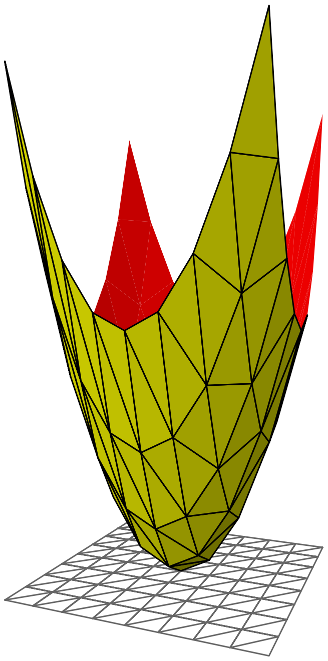


and continue on to examine all coordinates necessary to distinguish the points. Again we get a tree structure, one for which the equation nodes are coordinate equalities.

This can be used in a straightforward way to display lattice points:



But it can do something else.



The graph of the function $z = f(x, y)$ over a rectangle can be drawn if we first divide the rectangle into small rectangles, then divide each of the rectangles into two triangles, and plot the corresponding faces of the graph surface. But the graph will not necessarily be convex, and we must draw it back to front. Its structure is rather simple for this purpose. It can be drawn effectively in any position by applying the technique described above to construct a binary partition of space corresponding to the cylinders over the small rectangles in the xy plane.

Part 6. Miscellaneous

Stuff that didn't seem to fit elsewhere.

27. Vectors

The module `piscript.PiModule` contains a Python class `Vector`, which is also available separately in the module `piscript.Vectors`. These files define 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.x(v)`	the cross product (of 3D Vectors)
`u.length()`	the Euclidean length
`abs(u)`	also the Euclidean length
`u.normalized`	returns u scaled to length 1
`len(u)`	the length as an array—its dimension
`u.arg()`	the angular coordinate the 2D vector u, in radians
`u.rotate()`	*replaces* the coordinates of u by those of its rotation through a (in radians)
`u.rotated()`	returns a new rotated vector
`u.linethrough(v)`	returns $[A, B, C]$ defining the line through itself and v
`u.interpolated(v, t)`	returns the interpolation $(1-t)u+tv$

In those operations with two arguments (such as $u + v$), the first operand u must be a Vector, but the second operand v can be just a numerical 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 (all?) methods whenever coordinate arrays are intended.

The module `vectors` contains a similar set of commands with an extra initial argument u to replace the Vector in these. but whereas the methods in the Vector class return Vectors, those in `vectors` take arrays as arguments and return arrays. Thus:

```
import piscript.vectors as V

a = [1,2]
b = [2,3]
c = V.sum(a, b)
d = V.diff(a, b)
```

```
e = V.minus(a)
x = V.mul(a, b)
f = V.mul(a, 2)
```

There are also similarly related modules Matrices and matrices. The first defines a class Matrix with initializer a double array. The rows of the Matrix are vectors. The useful operations are multiplication `A*B`, `(Matrix).transform(v)`,and `(Matrix).transpose()`.

28. Addressing PostScript directly

There are a number of operations that interact more directly with the PostScript file. These include at the moment the only way to import images, such as photographs.

- `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. The other does no encapsulation, and is largely intended for loading fonts, where you want the imported file to affect the environment.

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. Then you would use `importEPS`.

```
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")
```

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 ...`

In the future, $\mathcal{PS}$ will be able to do operations on the images you import.

- `eol()`

Adds a blank line to the output PostScript 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 PostScript 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.

29. 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 in ΠS 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. 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.

30. 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.

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 tasks 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 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. If 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 **Mathematical 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 virtues of the packages that do figures from within TeX. It is certainly possible now that ΠS can handle virtually any TeX environment. I myself see no advantage to this compatibility, and for many reasons prefer fonts in figures to be definitely distinct from the fonts used for the same symbols in the main text. More specifically, I think that in text characters are designed to fit well together in one mass, whereas in figures the text stands isolated, and ought to be of heavier weight. I am well aware that emotions can run high over this matter.

Is it worthwhile for me 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.

Part 7. Coordinate systems

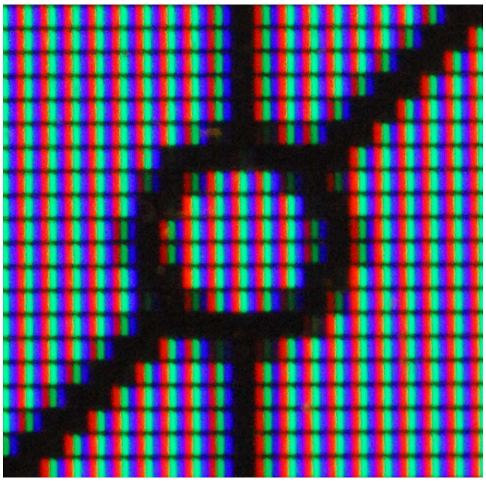
In order to use πS efficiently it is important to understand coordinate systems. This is standard fare in mathematics courses, but what is interesting about the way they are dealt with in both PostScript and πS is that internally they work with an extra dimension. Understanding how and why this is done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

31. Coordinates in 2D

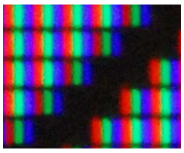
Making a figure with πS involves writing down a lot of instructions involving coordinates. But in order to see what your program actually produces, these instructions and numbers must be transformed at some point into instructions to a piece of hardware, for example either your computer display or through 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. First of all, each little is one pixel—that is to say one minimal unit in the image. Up close the separate RGB parts are visible, but from a distance, sure enough, they merge to form something close to white. Most interesting is probably 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. In other words, the image is smeared out a bit.



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 $(x_{\mathrm{pixel}}, y_{\mathrm{pixel}})$ in the window, where

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

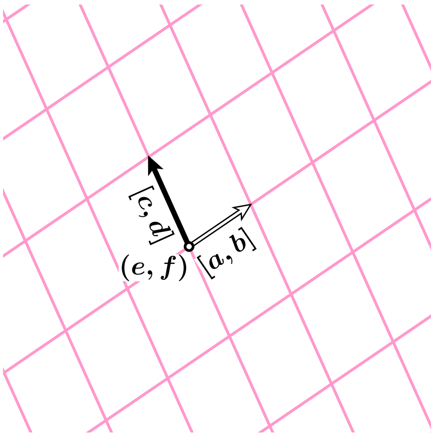
For example, $(0, 0)$ maps to $(0, 825)$ and since $825 = 11 \cdot 75$, this is indeed at lower left.

32. Affine transformations

The coordinate systems used in most graphics programs are **affine**. (I do not know the derivation of this term. My dictionary tells me that 'affine' is an English word coming from the Latin word 'affinis', which means 'related'. So much for that.) 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 every straight line in one is transformed to a straight line in the other. Algebraically, it means that the new coordinates (x, y) and the old ones (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 both PostScript and πS as a single array $[a, b, c, d, e, f]$. 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. 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 PS 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

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

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

33. The mathematics of coordinate changes

When PS 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 PS program proceeds, the coordinate system changes, and PS itself keeps track of the matrix transforming one set of coordinates to another. This matrix is part of the graphics state. *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

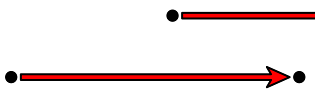
$$\begin{bmatrix} x_{\text{default}} \\ y_{\text{default}} \\ 1 \end{bmatrix} = T \begin{bmatrix} x_{\text{old}} \\ y_{\text{old}} \\ 1 \end{bmatrix} = TA \begin{bmatrix} x_{\text{new}} \\ y_{\text{new}} \\ 1 \end{bmatrix} .$$

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

34. 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.



Both positions and vectors intrinsic meaning, but given a coordinate system they'll also correspond to arrays of numbers. But the two notions, intrinsic meaning and coordinate expressions are not to be confused. As a very close analogy, the speed of an automobile is an intrinsic thing, but its specification as km/hour or mi/hour depends on a choice of scale—in effect a coordinate system.

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 relation between the two—to an affine function $ax + by + c$ is associated its **gradient** 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 obtaining an answer to the question, *How does a linear coordinate change affect a linear function?*

For example, suppose we change coordinates $x_{\mathrm{new}} = 2x_{\mathrm{old}}$, $y_{\mathrm{new}} = 3y_{\mathrm{old}}$. The point that is $(1, 1)$ in the original system becomes $(2, 3)$ in the new one. Suppose we now consider the line whose equation was originally $x_{\mathrm{old}} + y_{\mathrm{old}} = 1$. What is its in the new coordinates? Another way to pose this is to think, we have a fixed linear function, that is to say an assignment of a number to every point in the plane. In the old coordinate system its expression is $x_{\mathrm{old}} + y_{\mathrm{old}}$. What is its expression in the new one? The function doesn't change its value at a point, however, so its expression in the new system is $x_{\mathrm{new}}/2 + y_{\mathrm{new}}/3$.

You should think of a linear function as an intrinsic object that assigns numbers to vectors, and an affine function as one that assigns numbers to points. Both are very specila types of functions, with certain properties characterized as linear. But the formula for that function depends on a choice of coordinate system. How do we calculate the way this function changes when we change the coordinate system?

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

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

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

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

if we set

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

The same thing happens with affine functions.

Theorem. *If*

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

then the equation in new coordinates of the line $a_{\mathrm{old}} x_{\mathrm{old}} + b_{\mathrm{old}} y_{\mathrm{old}} + c_{\mathrm{old}} = 0$ *are obtained from*

$$\begin{bmatrix} a_{\mathrm{new}} & b_{\mathrm{new}} & c_{\mathrm{new}} \end{bmatrix} = \begin{bmatrix} a_{\mathrm{old}} & b_{\mathrm{old}} & c_{\mathrm{old}} \end{bmatrix} A \, .$$

If we identify 2D points with 3D ones, an interesting thing 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.

35. Homogeneous coordinates

An affine function $ax + by + c$ determines the line $ax + by + c = 0$. But what exactly are the coordinates of the line? Not (a, b, c), which can reasonably be considered the coordinates of the function, because 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, c) is not the same as $(-a, -b, -c)$. This also has some intrinsic meaning, because it allows us to distinguish sides of the line: $ax + by + c > 0$ and $ax + by + c < 0$. We can call a line together with a choice of sides an **oriented line**.

It also makes some sense to consider homogeneous coordinates for vectors, too: let the vector $[a, b]$ in 2D correspond to the 3D vector $[a, b, 0]$, and if we allow multiplication by a positive scalar we still preserve the **direction** of the vector. Two directions $[a, b]$ and $[a_*, b_*]$ are the same when one is obtained from the other by a positive scalar multiplication.

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

Earlier, we have mapped the 3D point (x, y, z) to the 4D point $(x, y, z, 1)$. But now I extend this, and map it to the whole collection of 4D points (cx, cy, cz, c) for $c > 0$—in other words, we are assigning it oriented homogeneous coordinates. Why do that? It turns out that when perspective, lighting, and visibility all have to be taken into account, as they do in 3D drawing, it is a very natural and efficient thing to do.

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

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

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:

Theorem. *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. The homogeneous coordinates of R are equal to*

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

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. Actually, even in this case the formula makes some sense, if we recall that 4D vectors with last coordinate 0 may be interpreted as directions.

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 PS 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, PS 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. Every 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

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

which can be written

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

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 8. Advice on illustrating mathematics

With πS you have at hand a tool for producing good mathematical pictures of a very high quality. It is not trivial to use well, but then producing useful mathematical figures is not a trivial task. I'd guess that πS is about as simple as it can be. But understanding the technical aspects of πS is just the beginning. The hardest part of producing good mathematical figures is the intellectual challenge involved, just as writing a good mathematical paper is much more than just being able to write words on a page (or key in letters on a compute). It requires imagination and care.

I offer here ten rules that I generally try to follow.

1. *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.

2. *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.

3. *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 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. One thing to be aware of is that mathematics writing often has two conflicting goals—one is to explain the basic idea(s) behind a proof, and the other is to give details of a rigourous proof. Pictures can be invaluable in the first role.

4. *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.

5. *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.

6. *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.

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

8. *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.

9. *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.

10. *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. I am pretty sure this is a common experience. Pictures can often be read rapidly, and adding more should always be taken as a option to be taken seriously. Also, although I myself sin badly in this respect, good captions are important.

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

In the original version of πS, the PostScript versions of TeX fonts were distributed along with πS itself. The new version, at least for Linux systems, uses the tools that come with the standard TeX distribution, if it has been installed.

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. I'll start here by assuming that all but πS is installed on your machine, and tell you later about auxiliary programs required.

36. 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
```

Unpack it in whatever you choose for your base directory. This will create and fill a directory `piscript-<version>` (which I'll refer to as `<base>`) with subdirectories `piscript`, `examples`, `docs`, `configs`.

The first contains the package of Python files, as well as a few programs written in C, that implement πS; the second contains a large collection of examples of πS programs and the `.eps` files they produce; the third contains this manual. The directory `configs` contains some sample `TexConfig` files. You should move these some place convenient. For example, on my machine I have moved `piscript` to be a subdirectory of `lib/python` in my home directory. (More precisely, I made a soft link with `ln -s <base>/piscript piscript` in `lib/python`.) In this document, I'll refer to this directory as `<piscript>`.

Now you must

(1) Register the directory `<piscript>` as part of your `PYTHONPATH` in your shell .

(2) Choose (or create) a directory somewhere in your personal home directory to be your `LOCALPISCRIPTDIR`, and register that also. I'll explain in a moment what it's for.

How to do (1) 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 `<base>` containing `piscript`—`$HOME/lib/python/`, for example.

As for (2), 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 `LOCALPISCRIPTDIR` to `$HOME/.piscript`. I also make an empty file `__init__.py` in `configs`, which makes it a possible home for a Python package. This directory will hold Python programs accessible to your own Python programs, no matter where they happen to be. The subdirectory `LOCALPISCRIPTDIR/configs` will be one searched automatically by πS to locate TeX configuration files.

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 LOCALPISCRIPTDIR='$HOME/.piscript'
export PYTHONPATH=$HOME/lib/python/:$HOME/.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.

37. Font access under UNIX

In the original distributions, I included a core selection of fonts and font data necessary to run most but not all TeX files. The current version, thanks to David Maxwell, will handle any TeX your machine will, as long as your TeX distribution is the standard one obtained from `CTAN`.

38. TeX environment

You might want to set the global TeX environment in the file `TexConfig.py`, which ought to be placed in the `<piscript>` directory. But if you use LaTeX in a standard way, you should be happy with the default. If you use plain TeX, you should copy `<piscript>/PlainTexConfig.py` to `<piscript>/TexConfig.py`. In time, you will probably want to customize your TeX environment. But this is easy to do later. How to do it is explained in the main text.

39. Installing under Windows

At the moment, a working version for Windows machines is not available. Real Soon Now.

40. History

For many years, I taught a course in geometry that used PostScript as a drawing tool. But this always involved a lot of work teaching just the basics of PostScript programming. Shortly after I retired from teaching, William Stein brought my attention to the programming language Python, and I realized very quickly that it would not be difficult to construct a Python interface to PostScript that would be much easier both to use and to teach.

The original version was tied very closely to PostScript, but it became clear that this was not necessary. Beginning in the summer of 2009, with the assistance of David Maxwell, I started converting the basic routines to eliminate this dependence. The eventual conclusion to this trend was to filter all output through the command arrays I refer to in the manual. This should have several interesting consequences in the future, among them (1) direct `.pdf` output and (2) being able to geometrically transform an arbitrary figure. Other current work in progress is concerned with creating practical BSP routines for drawing several 3D objects.

About the same time, Maxwell expanded the font-handling capabilities enormously, mostly by making it possible to deal with Don Knuth's virtual fonts, and by using the CTAN distribution's tools to locate font files. A number of other minor but exceedingly pleasant improvements, such as eliminating the need to specify the output file name in `init`, are due to him.

Earlier in the Spring of 2009, I started working out the idea of 'meta-graphics', but only finally came to `place` and `embed` a year later.

Much of the development of πS is greatly indebted to Günther Ziegler's renewed invitations to give a graphics workshop to graduate students at the Berlin Mathematical School for each of four summers 2007–2010.

41. Auxiliary programs

I shall 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/
```

I can also tell you that under no circumstances do you want to get Python 3, which is a very different language from the more relaxed and friendly Python 2.

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. It worked with earlier versions of πS, and will be the principal target for the present one.

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 (very) 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`

42. 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 a Python list (basically, a changeable array) of the integers $1, 2, \ldots, 99$. You can see this by

```
print range(10)
```

It is somewhat inefficient to construct the full range, and you can use a dynamic equivalent `xrange`. 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 one other characteristic feature of Python. 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.

43. 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 $qm \le n < (q+1)n$, 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 seems 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. Thus $x = 1.0 * a/b$ (probably incorrect, because a/b might be evaluated first) is not the same as $(1.0 * a)/b$ (probably what you want).

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).

Strings are arrays. You can refer to `len(s)` or `s[i]`. The last element is `s[-1]`,a nd you can get a substring as `s[3:6]` or `s[3:-1]`.

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 a list by appending data to it, and you can delete entries from it with `remove`. 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.

If you are never going to modify your array, use a tuple (such as `(1,2,3)`).

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 `None` object (equivalent of a null pointer in other languages) 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 `selfish`. 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.

44. 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 truly spectacular efficiency in C++ programs often relies on clever memory allocation.

45. 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.

46. Conditionals

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

Testing equality is done with ==.

47. 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.

In a πS program, once you have imported `PiModule` you get `math` for free.

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

48. 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`.

49. 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{images/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 first line says that extensions `.py` and `.eps` are "of interest". The next group says that in order to produce a file `x.eps` from a file `x.py`, Python should be run on it. 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. Index of commands

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

Basic commands

3D commands