Quick course in PostScript drawing

by Bill Casselman

These notes are intended to provide a basic short course in how to use PostScript for drawing mathematical figures. Many people, if they are familiar with PostScript at all, probably think of it as a primitive language produced by a computer drawing program like Adobe Illustrator for nothing but producing pictures on a printer. In fact, it is a well designed and versatile programming language with which you can produce almost any mathematical illustration you desire.

It is made more useful by the fact that there are freely available programs **Ghostscript** and **Ghostview** or **GSView** which translate your programs into screen pictures. They run on almost any machine you are likely to meet. You can download them for any platform from

`http://www.cs.wisc.edu/ ghost/`

Your principal source of information on generating mathematical will probably be

`http://www.math.ubc.ca/ cass/graphics/text/www/`

1. Starting

To draw a picture with PostScript, write a file of commands and then use `gsview` (or `ghostview`) to interpret it and draw a picture on the screen. You can keep modifying your file and reopening it from the same window until it looks right. If you want a printed copy you send the file to the printer through your viewer also.

Every PostScript file should begin with the 'magic' characters `%!`*. These should be the very first two characters of the file.*

Comments in PostScript are generally begun with `%`. The rest of the line after a `%` will be ignored when the computer reads the file. To get a file printed, your file should end with `showpage`. Spaces and line ends are all the same in PostScript.

2. Driving in reverse

The first feature of PostScript that causes some confusion is that commands and operators are expressed in an order somewhat backward from what you may be used to. To add 3 and 4 you type `3 4 add`. To find the sine of 40° you type `40 sin`. Operations and commands apply to a **stack**—an array of data—much like with an HP calculator. Operations are applied to things at the **top** of the stack, and usually remove them to replace them with the result of the operation.

This convention is called **RPN** (Reverse Polish Notation), after a group of mathematicians who invented it. Its virtue is speed of interpretation on a machine.

Arithmetic operations are in words: `add`, `sub`, `mul`, `div`. Mathematical functions available are `sqrt`, `sin`, `cos`, `tan`, `ln` (natural logarithm), `exp` (`y x exp` gives y^x), and `atan` (so that `y x atan` gives $\operatorname{atan}(y/x)$. All operations and commands leave the result at the bottom of the stack. Angles are in degrees.

Arrays in PostScript are bracketed by `[...]`, indexed starting with 0, accessed as in `[4 5 6] 1 get` which gives 5.

3. Drawing

PostScript is above all designed for drawing, but just as an army is designed for fighting and yet usually has only a relatively small number of fighting soldiers, PostScript has a small number of commands directly concerned with drawing.

Here is how a unit square is drawn:

```
newpath
0 0 moveto
1 0 lineto
1 1 lineto
0 1 lineto
0 0 lineto
stroke
```

The command `moveto` is like raising your pen from the paper and putting it down somewhere. The command `lineto` keeps the pen on the paper while it moves. Closely related commands are `rmoveto` and `rlineto`, which use coordinates **relative to the current position**. But *nothing is actually drawn until you enter* `stroke` *or* `fill`. The first draws a line along the path, the second fills in a closed path if you have drawn one. You can close a path up with `closepath`, always a good idea to include it if you mean to draw a truly closed curve.

Another drawing command is `arc`. Thus

```
newpath
1 2 0.5 0 360 arc
fill
```

fills in a circle with centre $(1, 2)$, radius 0.5. You can also restrict all subsequent drawing to within a path with `clip` (instead of `stroke` or `fill`).

The most interesting drawing command in PostScript is `curve`, but understanding how it works and how to use it is more complicated than I can explain here. Basically, it is used for drawing smooth curves which look good at all magnifications. For many purposes, however, you can approximate a smooth curve by a sequence of straight line segments.

4. The stack

You can't understand what your programs do without knowing a bit about how the stack works. The first basic fact is that most commands apply to items at the top of the stack, removing them and replacing them by the **return value** of the command. Thus the sequence `4 5 add` (1) puts 4 on the top of the stack; (2) next puts 5 on it; (3) removes these two items and returns by putting 9 on the stack, ready to be used in the next operation. The items a command swallows up are called its **arguments**.

Certain commands just manipulate items on the stack without changing them. The command `exch` exchanges the two items at the bottom (the exposed part) of the stack. The command `pop` gets rid of the bottom item on the stack. The command `==` displays and pops the top item. The command `dup` makes an extra copy of the item on the top of the stack (so there will now be two copies of it at the top).

5. Units

The natural unit of length in PostScript is one **point** which is $1/72''$ of one inch. The unit square in these units is barely visible. To change scale, for example to inches, type `72 72 scale`.

When PostScript begins, the width of lines is 1 unit. You can set it with the command `setlinewidth`. Thus `0.01 setlinewidth` is a good choice if you have scaled to inches. (If you do not reset the linewidth, it will remain at one unit—i.e. one inch.)

6. Coordinates

At the beginning of each program, the origin is at the lower left. But one thing that makes PostScript powerful is that you can change coordinate systems easily. You can shift the origin by `translate`. Thus

```
72 72 scale
4.25 5.5 translate
```

will put the origin at the centre of a $8.5'' \times 11''$ page.

You can rotate your figures with something like `40 rotate` (degrees again). Be sure to unrotate afterwards.

7. Colours

Black is 0, white is 1, grey in between. Thus `0.5 setgray` sets the current colour to medium grey (note American spelling). One figure always paints over an earlier one.

Using colours can be striking. They are described in RGB terms. Thus `0 0 1 setrgbcolor` makes the current colour equal to blue.

Colours, line widths, scale, the origin of the coordinate system are all part of the **graphics environment**.

8. Pages

You end a page with `showpage`. It is best to set up coordinates and sizes over again on each page. You can best do this if you begin every page with a `gsave` (saving the default graphics environment) and ending it with `grestore showpage` (restoring the original default environment).

9. Variables

Variables are defined with `def`. Thus `/x 3 def` sets $x = 3$. It is possible to make variables local, but in this crash course we'll skip that.

10. Procedures

A **procedure** in PostScript is a sequence of commands enclosed in `{...}`. They can be assigned to variables just like anything else. Entering this variable afterwards just replaces the variable by the sequence it stands for. Thus

```
/degrees {180 mul 3.14159 div} def
```

defines a procedure which will change radians to degrees, as in `3.14159 degrees`, which gives $180°$.

11. Loops

There are several ways to loop in PostScript. The simplest is `repeat`. Thus

```
10 {2 mul} repeat
```

will multiply a number on the stack by 2^{10}.

Another is `for`. At the beginning of every loop here the loop variable is placed on the stack. Thus

```
0
1 1 10 {
add
} for
```

puts the sum $1 + 2 + \cdots + 10$ on the stack. *Only integers should be used as loop variables.*

12. Brief command list

```
newpath
moveto
rmoveto
lineto
rlineto
stroke
fill
clip
closepath
arc
==
gsave
grestore
scale
translate
rotate
cos
sin
exp
ln
atan
add
sub
mul
div
sqrt
setlinewidth
setgray
setrgbcolor
showpage
for
get
aload pop
exch
def
repeat
dup
[ ... ]
```