Mathematics 308 — Geometry

Chapter 1. Getting started in PostScript

PostScript is a rather low level computer language developed by Adobe Systems. Its primary purpose is to produce high quality graphics from computers, and even to output computer graphics on printers. It is nonetheless a convenient computer language for doing geometry. In this course we shall use a program called **ghostscript**, as well as one of several programs which in turn call on **ghostscript**, to serve as our PostScript interpreter and interface. All the programs we shall use are available without cost through the Internet, as will be explained elsewhere.

The interpreter **ghostscript** has by itself a relatively primitive user interface which will turn out to be too awkward to use for very long, but learning this interface will give you a valuable feel for the way PostScript works. Furthermore, it will have a function later on when we produce simple animations. We shall begin in this chapter by showing how **ghostscript** works, and then later on explain a more convenient way to produce pictures with PostScript.

1. Simple drawing

Start up **ghostscript**. On the Mathematics network this is done by typing **gs**, and on other systems it is usually done by clicking on the icon for **ghostscript**. What you get when it starts up are two windows, one a kind of terminal window into which you type commands and from which you read plain text output, and the other a graphics window in which things are drawn.

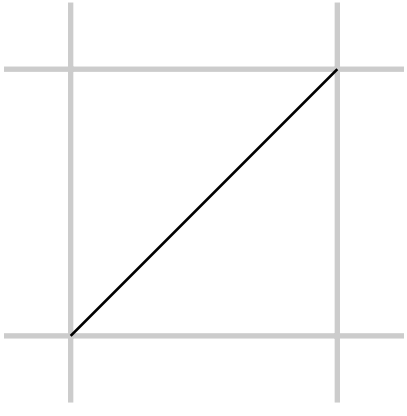
The graphics window—which I shall often call the **page**—opens up with a default coordinate system. When PostScript starts up, the origin of the coordinate system on a page is at the lower left, and the unit of measurement—which is the same in both horizontal and vertical directions—is equal to a **point**, exactly equal to $1/72$ of an inch. (This **Adobe point** is almost, but not quite, the same as the classical printer's point, which measures 72.27 to an inch.) Usually the size of the graphics window is $8\,1/2'' \times 11''$, or 612×792 points. As we shall see in a moment, the coordinate system can be easily changed so as to arrange x and y units to be anything you want, with the origin anywhere in the plane of the page.

When we start up running my local version of **ghostscript** in a terminal window we get a display like this:

```
Initializing...  done.
Ghostscript 2.6.1 (5/28/93)
Copyright (C) 1990-1993 Aladdin Enterprises, Menlo Park, CA.
  All rights reserved.
Ghostscript comes with NO WARRANTY: see the file COPYING for details.
GS>
```

In short, we are facing the **ghostscript prompt GS>**, and we are expected to type in commands. Let's start off by drawing a line in the middle of the page. Here is what the terminal window looks like when we do this:

```
GS>newpath
GS>300 400 moveto
GS>400 500 lineto
GS>stroke
GS>
```



The machine produces the prompts here, and everything else is typed by you. The graphics window displays the diagonal line in the figure on the right; I have added a grid to make the picture clearer.

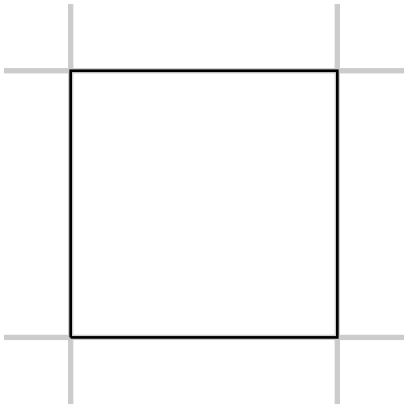
You draw things in PostScript by drawing **paths**. A path is a sequence of lines and curves. First you build a path, and then you actually draw it.

- You begin building a new path with the command **`newpath`**. This is like picking up a pen to begin drawing on a piece of paper.
- You start the path itself with the command **`moveto`**. This is like placing your pen at the beginning of your path. Things are generally what you might think to be backwards in PostScript, so you first write down the coordinates of the point to move to, then the command.
- You add a line to your path with the command **`lineto`**. This is like moving your pen on the paper. Again you place the coordinates first, then the command.
- So far you have just built your path. You draw it with the command **`stroke`**. You have some choice over what colour you can draw with, but the colour that is used by default is black.

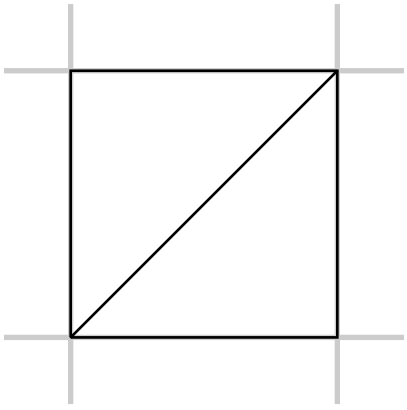
From now on I will usually leave the prompts out.

You would draw a square 100 points on a side with the command sequence

```
newpath
300 400 moveto
400 400 lineto
400 500 lineto
300 500 lineto
300 400 lineto
stroke
```



If you type this immediately after the previous command sequence, you will just put the square down on top of the line you have already drawn:



I will tell you later how to start over with a clean page. For now, it is important to remember that *PostScript paints over what you have already drawn*, just like painting on a canvas. There is no command that erases stuff already drawn.

There are often lots of different ways to do thing in PostScript. Here is a different way to draw the square:

```
newpath
300 400 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
closepath
stroke
```

Here **rmoveto** and **rlineto** mean motion **relative** to where you were before. The command **closepath** closes up your path back to the last point to which you applied a **moveto** point.

A very different effect is obtained with:

```
newpath
300 400 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
closepath
fill
```

This just makes a big black square in the same location. Whenever you build a path, the operations you perform to make it visible are **stroke** and **fill**. The first makes an outline, the second fills the inside of the path.

You can draw in colour with two commands, **setgray** and **setrgbcolor**:

```
0.5 setgray
newpath
300 400 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
closepath
fill
```

will make a grey square, and

```
1 0 0 setrgbcolor
newpath
300 400 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
closepath
fill
```

will make a red one. The **rgb** here stands for Red, Green, Blue. Whenever you have set a new colour, it will generally persist until you change it again. Note that **0** is black, **1** white. The command **x setgray** is the same as **x x x setrgbcolor**. You can remember that **1** is white by recalling from high school physics that white is made up of all the colors put together.

If you want to draw a red square with a black outline you type

```
1 0 0 setrgbcolor
newpath
300 400 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
closepath
fill
0 setgray
newpath
300 400 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
closepath
stroke
```

Exercise 1.1. *Run* **`ghostscript`** *and draw an equilateral triangle near the centre of the page, instead of a square. Make it* 100 *points on a side. First draw it in outline, then fill it in black. Next, make it in turn red, green, and blue with a black outline. (You will have to do some calculations first.)*

2. Simple coordinate changes

Working with points as a unit of measure is not for most purposes very convenient. For North Americans, since the default page size is $11'' \times 8.5''$, working with inches usually proves easier. We can change the basic unit of length by typing

```
72 72 scale
```

which scales up the x and y units by a factor of 72. Scaling affects the current units, so scaling by 72 is the same as scaling first by 8, then by 9. This is the way it always works:

- *Coordinate changes always affect the current coordinate system.*

When you scale, you must take into account the fact that the default choice of the width of lines is 1 unit. So if you scale to inches, you will get lines 1 inch wide unless you do something about it. It might be a good idea to add

```
0.01 setlinewidth
```

when you scale to inches. This sets the width of lines to a hundredth of an inch. A linewidth of 0 is also allowable—it just produces the thinnest possible lines which do not actually vanish.

Exercise 2.1. *How would you scale to centimeters?*

You can also shift the origin.

```
1 2 translate
```

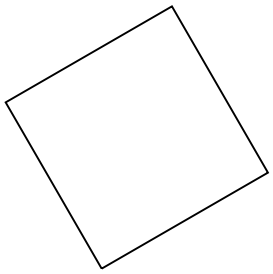
moves the coordinate origin to the right by 1 unit and up by 2 units. The combination

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

moves the origin to the centre of the page.

One more simple coordinate change rotates things.

```
72 72 scale
0.01 setlinewidth
4.5 5.5 translate
30 rotate
newpath
1 0 rlineto
0 1 rlineto
-1 0 rlineto
closepath
stroke
```



Note that *PostScript works with angles in* **degrees**. This will cause us some trouble later on, but for now it is probably A Good Thing.

3. Doing arithmetic in PostScript

PostScript is a complete programming language. It has limited arithmetical capabilities. As with drawing, the sequence of commands is backwards from what you might expect. To add two numbers, first enter the numbers, followed by the command **add**. The result of adding numbers is also not quite what you might expect. Here is a sample run:

```
GS>3 4 add
GS<1>
```

What's going on here? What does the **<1>** mean? Where is the answer?

PostScript uses a **stack** to do its operations. This is an array of arbitrary length which grows and shrinks as programs move along. The first item entered is said to be at the **bottom** of the stack, and the last item entered in is said to be at its **top**. This is rather like manipulating dishes at a cafeteria. Generally, operations affect only the things towards the top of the stack, and compute them without displaying results. For example, the sequence **3 4 add** does this:

Entry	*What happens*	*What the stack looks like*
3	The number 3 goes onto the bottom of the stack	**3**
4	The number 4 goes above the 3 on the stack	**3 4**
add	The operation **add** goes above 4 . . .	**3 4 add**
	. . . then collapses the stack to just a single number	**7**

You might be able to guess now that the **<1>** in our run tells us the size of the stack. To display the top of the stack we type **=**. If we do this we get

```
GS>3 4 add
GS<1>=
7
GS>
```

Note that **=** removes the result when it displays it (as does the similar command **==**). An alternative is to type **stack**, which displays the entire stack, and does not destroy anything on it.

```
GS>3 4 add
GS<1>stack
7
GS<1>
```

Other arithmetic operations are **sub**, **mul**, **div**. Some of the mathematical functions we can use are **sqrt**, **cos**, **sin**, **atan**. For example, here is a command sequence computing $\sqrt{3*3+4*4}$.

```
GS>3 3 mul
GS<1>4 4 mul
GS<2>add
GS<1>sqrt
GS<1>=
5.0
GS>
```

Exercise 3.1. *Display the stack as this calculation proceeds.*

Exercise 3.2. *Use* **ghostscript** *to calculate* $\sqrt{9^2+7^2}$.

Here is a list of nearly all the mathematical operations and functions.

x y add	puts $x+y$ on the stack
x y sub	puts $x-y$ on the stack
x y mul	puts xy on the stack
x y div	puts x/y on the stack
x neg	puts $-x$ on the stack
y x atan	puts the polar angle of (x,y) on the stack (in degrees)
x sqrt	puts $\sqrt{x}$ on the stack
x sin	puts $\sin x$ on the stack (x in degrees)
x cos	puts $\cos x$ on the stack (x in degrees)
y x exp	puts y^x on the stack
x ln	puts $\ln x$ on the stack
x abs	puts $\lvert x\rvert$ on the stack

Exercise 3.3. *Use* **ghostscript** *to find* $\mathrm{acos}(0.4)$. *(This will require thinking a bit about the geometry of angles.) Recall that* $\mathrm{acos}(x)$ *is the unique angle between* 0° *and* 180° *whose cosine is* x.

4. Errors

You will make mistakes from time to time. Unfortunately, the way errors are handled in **ghostscript** (and indeed in all PostScript interpreters I ma familiar with) is very poor. This is not an easy problem to correct, unfortunately. Here is a typical session with a mistake signaled. If I enter

```
GS>5 0 div
```

this is what I get spilled out on the screen:

```
Error:  /undefinedresult in --div--
Operand stack:
   5   0
Execution stack:
   %interp_exit   --nostringval--
     --nostringval--   --nostringval--
     %loop_continue   --nostringval--
     --nostringval--   false   --nostringval--
```

```
    --nostringval--   --nostringval--
Dictionary stack:
  --dict:592/631--   --dict:0/20--   --dict:34/200--
Current allocation mode is local
GS<2>
```

The real point of this and just about all error messages from **ghostscript** is that you can ignore all but these first lines:

```
Error:  /undefinedresult in --div--
Operand stack:
  5   0
```

which shows you the general category of error and what the stack was like when the error occurred. Here it is division by 0. *It never pays to try too hard to interpret* **ghostscript** *error messages.* The only way to deal with them is to try to figure out where the error occurred, and examine your input carefully. There is trick you can use to find out where the error occurred: put lines like **(1) =** or **(location #1) =** at various points in your program and try to trace how things go from the way output is displayed. Simple, but often it helps. The way this works is that **(1)** denotes the string "1", and **=** will display it on the terminal. If you are running **ghostscript**, then you can clear the stack completely and start over with the command **clear**.

Incidentally, the way errors are handled by your PostScript interpreter can be modified by suitable embedded PostScript code. In particular, there is an apparently convenient error handler called **ehandler.ps** available from Adobe via the Internet, at **www.adobe.com**. If you have a copy of it in your current directory, you can use it by putting

```
(ehandler.ps) run
```

at the top of your file. You can also arrange for **ghostscript** to use it instead of its default error handling, but exactly how depends on which computer you are using.

5. Working with files and ghostview or GSView

Using the **ghostscript** interpreter directly shows interesting things, and you should be able to do it occasionally, but it is an extremely inefficient way to produce pictures, mostly because data entered cannot be changed easily, and errors will usually force you to start all over again. What is much better is a procedure with these components to it (**gview** is either **ghostview** or **GSView** depending on you system):

- *Start up* **gview**.
- *Start up a text editor.*
- *Create or open up in your text editor the file you want to hold your PostScript program. Be sure your file is to be saved as plain text, as opposed to one of the special formats word processors like to save. On Windows machines, this is the default with the simple editing program* **Notepad**.
- *Open up that file from* **gview**.
- *As you make up your program inside the editor, save it from time to time and reopen it in* **gview**, *where your picture and possibly other messages will be displayed.*

There are some new features of using files for PostScript programs that you'll have to take into account, but otherwise this works well, almost painlessly.

- *At the very beginning of your file you must have the two characters* **%!**. *This tells your computer that the file is a PostScript file.*
- *At the end of your file you should have a line with* **showpage** *on it.*

Neither of these is usually absolutely necessary, but there will be times when both are required. They will definitely be required if you want to print out your picture on a printer.

The command **showpage** displays the current page and then starts a new page. Later on you will want to make up files with several pages in them, and each page must have a **showpage** at the end. There is one tricky feature of **showpage**, however.

- *Scaling must be done over again on each page.*

There are better and worse ways to deal with this. The best is to put the commands **gsave** at the beginning and **grestore** at the end of each page. We will see later exactly what these commands do. Here, for example, is a complete two-page program:

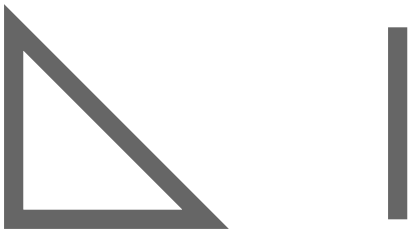
```
%!

gsave

72 72 scale
0.01 setlinewidth
4 5 translate

newpath
0 0 moveto
1 0 lineto
0.5 1 lineto
closepath
stroke

grestore

showpage

gsave

72 72 scale
0.01 setlinewidth
4 5 translate

newpath
0 0 moveto
1 0 lineto
1 1 lineto
closepath
stroke

grestore

showpage
```

Exercise 5.1. *What does this program do?*

6. Some fine points

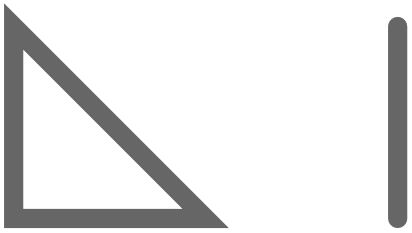
Here are a few lines that produced the figure on the right, after scaling to inches.

```
0.1 setlinewidth
newpath
0 0 moveto
1 0 lineto
0 1 lineto
closepath
stroke
```



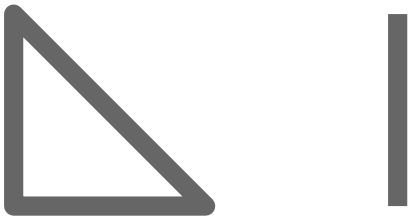
If I add a single line like the following, I get something very slightly different.

```
1 setlinecap
```

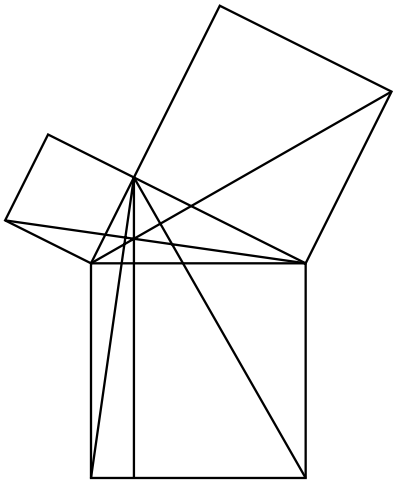


Another possibility:

```
1 setlinejoin
```



Exercise 6.1. *Draw in PostScript the following picture, taken from the proof of Proposition I.47 in the standard edition of Euclid's Elements. Use colours to help explain Euclid's argument.*



Here and elsewhere, when you are asked to reproduce a picture, you are almost always expected to reproduce its dimensions as closely as you can.

Exercise 6.2. *Draw a picture of the French flag (blue, white, and red vertical stripes), with thin black lines separating the different colours and outlining the flag. I do not know the official aspect ratio.*