PostScript Drawing:
An Economist's Guide

Alan G. Isaac
Department of Economics
American University
Washington, DC 20016
aisaac@american.edu

April 15, 2003

Abstract

Economists like to support their arguments with drawings. When these drawings are intended to illustrate very precise geometrical relationships, the PostScript page description language is an appropriate tool. It is easy to learn enough PostScript to gain a powerful, flexible tool for drawing. A few quick lines of PostScript can often replace frustrating and tedious experimentation in a "point and click" environment. This paper is an economist's introduction to PostScript drawing.

Contents

1 Background **4**
1.1 Essentials . . . 5

2 Getting Started **5**
2.1 Application . . . 6
2.2 Program Details . . . 6
2.2.1 Path Construction . . . 6
2.2.2 Painting . . . 7
2.3 Key Lessons . . . 7
2.3.1 Relative Movement . . . 8

3 Transformations **8**
3.1 Translate . . . 8
3.2 Scale . . . 9
3.3 Rotate . . . 9
3.4 Application . . . 9
3.5 Program Details . . . 10
3.5.1 Color . . . 10
3.5.2 Graphics State . . . 11
3.5.3 Line by Line . . . 11

4 Circles and Rectangles **11**
4.1 Drawing Rectangles . . . 11
4.2 Drawing Circles . . . 12
4.3 Application . . . 12
4.4 Program Details . . . 13
4.4.1 Variables . . . 13
4.4.2 Basic Math . . . 13
4.4.3 Line Width . . . 13
4.4.4 Line by Line . . . 14

5 Adding Text to a Drawing **14**
5.1 Choose Your Font . . . 15
5.2 Placing Text . . . 15
5.3 "Printing" Text . . . 16
5.3.1 Aligning Text . . . 16
5.4 The Symbol Font . . . 16

6 Clipping and Pattern Fills **17**
6.1 Dashed Lines . . . 17
6.2 Clipping Path . . . 18
6.3 Pattern Fills . . . 18
6.3.1 Hatching . . . 19
6.3.2 Stippling . . . 20
6.3.3 Just for Fun . . . 20

7 Encapsulated PostScript **21**

8 Conclusion **23**

A Programming in PostScript **25**
A.1 Postfix Notation and the Stack . . . 25
A.1.1 Some Useful PostScript Operators . . . 26
A.2 Procedures . . . 27
A.2.1 Named Procedures . . . 27
A.3 Flow Control . . . 27
A.3.1 Branching . . . 27
A.3.2 Looping . . . 28
A.4 Variables . . . 28
A.4.1 Scope . . . 28
A.5 Application: Drawing Arrows . . . 29
A.5.1 Curves . . . 29
A.5.2 Arrow Code . . . 30

PostScript Drawing:
An Economist's Guide

Economists like to support their arguments with drawings. Occasionally these drawings are intended to illustrate very precise relationships. Naturally this raises the question: what is the proper drawing tool for such occasions? Ideally the tool will be easy to learn yet powerful, allowing flexibility in the illustration of precise geometric relationships. The PostScript page description language meets these criteria.

This paper is an introduction to PostScript drawing. In contrast with the typographical emphasis of most PostScript introductions, it emphasizes a small set of simple PostScript language features that are particularly useful for the production of simple but precise drawings. The paper offers a self-contained introduction to PostScript drawing that is an adequate starting point for those who wish to add PostScript drawings to documents such as theses, dissertations, academic books, and journal articles. It also briefly discusses some more advanced topics and suggests resources for further study.

I began this paper as a guide for my students, in response to queries about how to add precise drawings to their theses, dissertations, and research papers. Naturally I first encouraged them to consider vector drawing programs. When the need for geometric precision or complex geometric shapes ruled out the easy use of vector drawing programs, I had been guiding my students to the extremely accessible LaTeX picture environment. Occasionally, however, they needed a more powerful drawing language.[1] PostScript provides this power, and learning to draw in PostScript requires surprisingly low effort. One can very quickly learn to produce complex but precise drawings in the powerful PostScript page description language—nearly as quickly as one can learn the LaTeX picture environment and more quickly than one can learn precise drawing in many "point and click" environments.

1 Background

In 1985, Adobe introduced the PostScript page description language. It soon became extremely popular for the production of professional drawings. For high resolution drawings intended for the inclusion in other documents, including academic books and journal articles, it quickly became the standard.

PostScript output is produced by many software applications. Many spreadsheets, data-analysis packages, and scientific programming languages allow the creation of sophisticated data-driven or even functionally driven graphics that, as a rule, can be exported as PostScript.[2] As a result, very few people have cause to turn write PostScript programs to create data-driven graphics.[3] There is also a plethora of drawing applications that support the export of PostScript graphics, and many of these applications require only a little "point and click" experimentation before the user can produce quite sophisticated drawings.[4] Why then would anyone turn to the underlying PostScript language to produce a drawing? Often there will be no reason: the existing applications are fully adequate for many drawing purposes. Nevertheless, three basic reasons frequently arise: precision, speed, and fun. A fourth reason arises occasionally and can be crucial when it does arise: it may prove necessary to tinker with the PostScript produced by an graphics application—possibly to alter line characteristics or object placement—and even a novice's understanding of PostScript often makes this possible.

PostScript programming should therefore been seen as a complement rather than a substitute for other approaches to the generation of vector graphics. The more precisely one wishes to render a simple drawing,

[1]The LaTeX picture environment is documented by Lamport (1994, appendix C.14.1). It offers a powerful mechanism for the placement of mathematics on a drawing. The LaTeX drawing constructs are useful but very limited when compared with PostScript; LaTeX drawings are also less portable (unless first converted to PostScript). If use of LaTeX is a given, the advantage of LaTeX in typesetting mathematics is easily combined with the power of PostScript drawing: just `\put{}` an included EPS graphic into a LaTeX picture environment, be careful about your units (PostScript points are the same as TeX big points), and then `\put{}` the math into the picture environment as well. (For a more elegant solution, see the PSfrag package.)

[2]For example, gnuplot is a free scientific plotting program that produces very flexible and high quality PostScript output.

[3]For those that do, Kunkel (1990) provides an excellent introduction.

[4]Widely used vector graphics applications include Xfig, Jfig, Mayura Draw, Xara X, Corel Designer, CorelDraw, and Adobe Illustrator.

the more likely it is that turning directly to the PostScript language will prove valuable. PostScript excels at the precise representation of geometric relationships. Since PostScript is a page description language that is particularly simple in structure, precise geometrical relationships that can be difficult to produce accurately in a point and click environment often require only a few minutes of PostScript programming. (As simple examples, compare the ease of production of figure 2 or figure 3 below with the difficulties in achieving precise equivalents in your favorite "point-and-click" vector graphics application, or consider the ease with which the techniques used in figure 2 can be generalized to the production of regular polygons of any dimension.) Finally, a little PostScript programming can produce results that are visually quite exciting, so it is impossible to neglect the fun involved.

1.1 Essentials

While vector drawing applications can be expensive, tools for PostScript programming are available for free. However the true cost of producing any drawing will have little to do with monetary cost, since drawing can be very time consuming no matter what environment is chosen. For most economists, drawing by means of PostScript programming will be worthwhile only in those circumstances where simple programs can produce results that would require time-consuming point-and-click experimentation in a vector graphics application. After dealing with a few essentials and introducing some basic considerations, this paper will provide a few examples of such programs.

To create and view PostScript drawings on a computer, you need only a text editor and an "interpreter" supporting screen display.[5] The inexpensive options for interpreters that permit onscreen viewing are limited but high quality. The standard is the free GhostScript interpreter, which allows previewing and printing PostScript files on many platforms.[6] As for text editors, this paper focuses on very simple examples, for which you can use the default editor on your computer (or even a word processor, if you remember to save what you type as ASCII text).[7]

Finally, for anyone intending to go beyond the most casual use of PostScript, I strongly recommend downloading Adobe 1999a, which is a precise yet usually readable reference manual.

2 Getting Started

This paper discusses the creation of PostScript drawings that fit on a single page. There are two basic steps in creating a PostScript drawing: first *construct* a **path**, then *paint* it. Almost all our effort in this paper will be spent on the first step.

The drawings in this paper are produced by extremely simple PostScript programs: ASCII text files containing only a few lines of PostScript code. Each program begins with a single line containing the two characters `%!`.[8] The PostScript program follows this special comment. Each program ends with the PostScript operator **showpage**.

PostScript allows easy construction of a line segment between any two points on the page: one just needs to specify the coordinates of the endpoints of the line segment. We can use the **moveto** operator to move to one endpoint, and then we can use the **lineto** operator to construct a line segment to the other endpoint. It follows immediately that you can easily construct any figure composed of straight line segments,

[5] Many printers include a PostScript interpreter (or a clone), but even people with access to such a printer will find it helpful to be able to view their drawings on screen.

[6] Many GhostScript users will also want a graphical interface to GhostScript, such as GSView (for Windows/Linux/OS2), MacGSView for the Mac OS, and GhostView for Unix and VMS. Windows users may be interested in the small, fast RoPS interpreter, which has a free LanguageLevel 1 version. Serious PostScript programmers may profit from PSAlter's helpful debugging facilities.

[7] Windows operating systems include NotePad, Unix and related systems usually include vi, and Macintosh operating systems usually include TeachText. These will be adequate for our simple experiments.

[8] This is not required by the PostScript language specification, nor is it required for rendering the examples in this paper using GhostScript. However, many applications, printers, and printer controllers recognize this as a special comment indicating that they are dealing with a PostScript file. For example, a printer with PostScript capability may rely on this special comment to determine whether to load a PostScript interpreter or a PCL interpreter. The PCL interpreter would print the source code rather than our drawing.

as long as you know the coordinates of the endpoints of each line segment. The ability to draw straight lines encompasses a surprising number of drawing needs. (Even the plots of non-linear functions are usually drawn as a series of short straight line segments.) Here we will explore the details with an extremely simple drawing: a right triangle. In order to emphasize the precision of PostScript drawings, we will be very specific about certain measurements.

2.1 Application

A classic first project in PostScript drawing is a simple triangle. We will draw a right triangle with two 6-inch sides. The basic measurement unit in PostScript is the **point**, which is 1/72 inches, so each side will measure 432 points. We will place the vertex of the right angle one inch (i.e., 72 points) from the bottom and one inch from the left side of our sheet of paper.[9] The following code accomplishes this project.

```
%!                    %declare file to be PostScript
72 72 moveto          %set starting point for path
504 72 lineto         %construct first line segment
72 504 lineto         %construct second line segment
closepath             %construct final line segment
stroke                %paint the constructed triangle
showpage              %end program with ‘showpage’
```

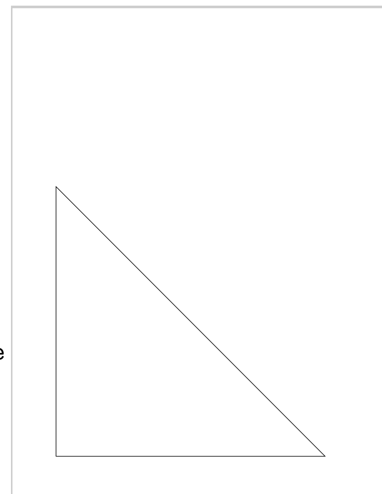

Figure 1: Line Segments

Begin this project by creating a new ASCII text file with any text editor. We might as well call it `first.ps`, since it will contain our first PostScript program. Next we enter the program code listed above. Recall that we want to construct a triangular path, and then paint it. In this example, we begin our path construction with the instructions `72 72 moveto`, which sets the beginning of the path to the point (72,72). Next we construct two line segments: the horizontal side with `504 72 lineto`, and the hypotenuse with `72 504 lineto`. We construct the vertical side in a special way: we use the **closepath** operator. We paint the constructed triangle with the **stroke** operator. Finally, we end our program with the **showpage** operator, and save the program file to disk. Use your PostScript interpreter to view this drawing:[10] when you open the `first.ps` file you will see the triangle in figure 1. Congratulations. You have produced your first PostScript drawing!

2.2 Program Details

We will now consider our first PostScript program in more detail. As mentioned earlier, the first program line is simply a special comment that declares the file to be a PostScript program file, and we end our program with the **showpage** operator. Between the first and last lines we find the two basic steps in creating a PostScript drawing: we first *construct* a **path**, and we then *paint* it. Since we have already briefly discussed these steps, you will have noticed how simple and intuitive the syntax and operator names are.

2.2.1 Path Construction

To begin constructing a path, we need to pick a point where we will start drawing. Points on the page are designated in a familiar fashion: we use standard Cartesian coordinates, with the position (0,0) at the lower

[9]For simplicity we are currently discussing only the location of the constructed path and not the width of the line when we stroke it.

[10]For example, if you are using GhostScript under Windows, just enter **(c:\\first.ps)run** at the command prompt (if that is the file location you chose).

left corner of the page. We now need to make an arbitrary choice: we can start constructing our path at any vertex of the triangle. In this example, we pick (72,72) as the starting position for the path we wish to construct, and we get there with the **moveto** operator. So the second line of our program is `72 72 moveto`, which instructs the interpreter to make the position (72,72) the current point of the path we are constructing.

The next step is to construct a line segment from (72,72) to another position on the page, for which we use the **lineto** operator. Our first line segment will be the horizontal side, which is 6 inches long. Recall that 6 inches is equal to 432 points, so we want our line segment to end at the position (504,72). Therefore the third line of our program is `504 72 lineto`. This instructs the interpreter to construct a line segment from the current point to the point (504,72), which then becomes the new current point of the path we are constructing.

So far, we have constructed a line segment from (72,72) to (504,72), which becomes the new current point. We still need to add two more line segments to our path. The fourth line of the program instructs the interpreter to construct a line segment from the current point to (72,504), which then becomes the new current point of the path we are constructing. In the fifth program line we complete our construction of the triangle: the **closepath** operator constructs a line segment from the current point of our path to the starting point of our path (as determined by our **moveto** command). In our case, the current point when we give the **closepath** command is (72,504), and we began our path at (72,72). This completes the construction of our triangle, and we are ready to paint it.

2.2.2 Painting

Constructing a path does *not* in itself give us anything to view. We need to "paint" the path to produce a drawing. The sixth line is the painting operation, for which we use the **stroke** operator. The **stroke** operator may be thought of as putting "ink" on the path we have constructed: until we **stroke** our path, there is nothing to view.

A natural question arises at this point: what determines the color and line width of the stroked path? For our first simple program, we simply used the PostScript defaults. The default color is black, and the default line width is 1 point. Sections 3.5.1 and 6 show how these can easily be changed.

Note that all of our path construction and painting code could have been placed on a single line. Instead we placed an easily interpretable code snippet on each line, and we included a helpful comment with each code snippet. This makes our code more readable and easier to debug.

The last program line is **showpage**, which instructs the interpreter to show us everything we have painted on the page.[11] Once we construct the path, **stroke** it, and use **showpage** to show what we have drawn, we can view the triangle illustrated in figure 1.

What if we had wanted a filled triangle rather than a stroked triangle? We need change only one line of our PostScript program: change the **stroke** operator to **fill** and view the new result. (Try it.) This fills our triangle with black. We can fill our triangle with other colors (or even patterns): see sections 3.5.1 and 6.

2.3 Key Lessons

Much of what we need to know in order to draw with PostScript is contained in this first simple program. Note in particular our use of units of measurement, a coordinate system, and postfix syntax. The basic unit of measurement is one point.[12] There are exactly 72 points per inch. So an 8.5 inch by 11 inch sheet of paper is 612 points by 792 points. The coordinate system is a standard Cartesian coordinate system, with the origin in the lower left corner of the page. The location (72,504), for example, is one inch (72 points) to the right of the origin and 7 inches (504 points) above it.

As for the syntax, you probably noticed that we always state a position *before* doing something with it (e.g., moving to it, or constructing a line segment to it). More generally, the PostScript interpreter should always encounter an operator *after* the operands required by that operator. For this reason, we say PostScript uses a *postfix* notation: an operator *follows* its operands. First you state the operands; then you

[11] The **showpage** operator also tells a PostScript printer to eject the page.

[12] This is a unit of measurement. Do not confuse it with the "points" of the Cartesian coordinate system, which are dimensionless positions.

state operator that will use these operands. We say that you *push* the operands onto the operand **stack**, where they can be found by the operator.[13] PostScript syntax is discussed in additional detail in section A.

Readers with programming experience will have noticed that we do not compile a PostScript program into an executable file that can run on a computer without an interpreter. We always use a PostScript interpreter to run our programs. We say that PostScript is an interpreted language.

We have already acquired enough tools to construct any drawing whose constituents are line segments—including graph axes, arbitrary polygons, linear functions, or even line graphs of time series—as long as we can specify the endpoints of each line segment. This drives home the flexibility of the PostScript page description language: after a few minutes of exposure, one is ready to create very complex drawings with great precision. This simplicity has another advantage: most vector drawing programs rely on proprietary and undocumented formats, whereas your own PostScript drawings will rely on a fully documented open standard. Of course one can usually export PostScript from a drawing application, but that PostScript is often filled with obscurities. Hand coded PostScript drawings will generally be easy to understand, easy to modify, and readily transportable.

2.3.1 Relative Movement

We have explore the use of the **moveto** and **lineto** operators in path construction. PostScript also provides the **rmoveto** and **rlineto** operators, which allow for movements relative to the current point rather than to absolute positions. The operands of the **rmoveto** and **rlineto** operators are not positions but are rather the *displacements* (dx,dy) relative to the current point. Once a current point is set we can use the **rmoveto** operator to make a relative movement (dx,dy) to the beginning of a new **path**subpath. Similarly the **rlineto** operator also takes as operands the horizontal and vertical displacements (dx,dy) relative to the current point. For example, we could replace the line `504 72 lineto` in our example program with the line `432 0 rlineto`. Similarly we could replace the line `72 504 lineto` with the line `-432 432 rlineto`. (Try it!) Reliance on relative movements can be particularly useful if we wish to construct the same object at multiple positions on the page: using relative movements allows us to reuse our code at various positions.

3 Transformations

Our example PostScript programs have illustrated the very natural coordinate system PostScript provides for describing positions on the page. It is a standard Cartesian coordinate system, with the origin at the bottom left corner of the page. The coordinate system in which the programmer is drawing is called the current user space, so up to now we have been drawing in the default user space. PostScript allows for very general transformation of the user space). In this section we will see how this can be useful.

This section discusses three operators for simple coordinate system transformations. These are **translate**, **scale**, and **rotate**. These operators effectively allow us to move the axes around on the page: we can shift them horizontally or vertically, we can "stretch" or shrink them (so that the axes are not moved but their units are changed), or we can rotate them around the current origin. These operators can be invoked even in the middle of the construction of a path, and they will only affect the construction subsequent to the invocation. For example, transformations do not move the location *on the page* of the current point or the current **path**. This section discusses the effects of these transformations and offers a simple but useful illustration.

3.1 Translate

The **translate** operator changes the location of the user space origin. The origin can be translated along both axes (i.e., both horizontally and vertically relative to the current user space). Consider the location on the page associated with a point (x, y) in the current user space. The instructions x y **translate** will move

[13] Many readers will be familiar with postfix syntax from the use of calculators that rely on reverse polish notation. I offer the online calculator PSCalc as a quick way to get familiar with some basic PostScript operators.

the origin to that location, so that that location on the page now has the coordinates $(0, 0)$ in the new user space). The two operands specify the x-axis translation the y-axis translation.

3.2 Scale

The **scale** operator allows us to change the units on the coordinate axes without affecting the origin or the orientation of the axes. The command s_x s_y **scale** imposes a "horizontal" scale factor of s_x and a "vertical" scale factor of s_y. For example, the location on the page that had coordinates (x_0, y_0) will, subsequent to the scale operation, have coordinates $(x_0/s_x, y_0/s_y)$.

Clearly order matters when using **scale** and **translate**. If scale comes first, the new units will be used in translating the origin of the user coordinate system.

3.3 Rotate

We can also **rotate** the axes without moving the origin or changing the scale. Use the command θ **rotate** to rotate the axes θ degrees counterclockwise.

3.4 Application

In section 2 we drew a right triangle. That illustrated how we can easily draw any figure consisting of straight line segments as long as we can specify the endpoints of each segment. Sometimes such specification can be considerable work, however, For example, instead of a right triangle we may wish to draw an equilateral triangle. The equilateral triangle is a common representation of the two dimensional simplex, so it shows up in a diversity of contexts.[14] But suppose we draw our first triangle side exactly as in our first example; how would we draw the second side? The brute force way is to trigonometrically compute the location of the second vertex and then draw a line segment to that point.[15] A simpler solution is to recognize that any triangle is determined by a side, an angle, and a second side. So after drawing the first side, we can translate the origin to the endpoint of the first line segment, rotate around the new origin by 120°, and then draw the second side exactly like the first one. We will write the code to do this so that it makes use of our three transformation operators. To illustrate a few additional points, we will both stroke and fill the constructed path. (In order to see the difference between the filled and stroked areas, we will use a different color for the fill.)

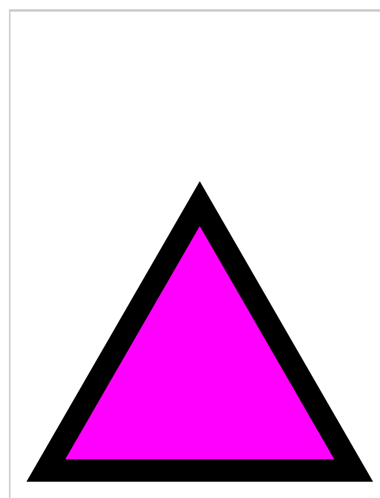

Figure 2: Equilateral Triangle

```
%!                       %declare file to be PostScript
72 72 scale              %set units to inches
1.25 1 translate         %translate origin
0 0 moveto               %moveto origin
6 0 lineto               %construct first line segment
6 0 translate            %translate origin
120 rotate               %rotate axes by 120 degrees
```

[14]Economic examples include the illustration of sectoral shares, the representation of strategy sets, illustration of the stability properties of tatonnement, representation of the allocation probabilities for an indivisible good, game theoretic proofs, and proofs related to social choice, public choice, and mechanism design. For randomly chosen illustrative examples from textbooks and journal articles see Carter (2001, ch.1), Hildenbrand and Kirman (1988, section 6.5), Karni and Safra (2002), Laffont (1988, ch.2), Moulin (2000, Appendix), or von Stengel et al. (2002).

[15]Since PostScript is a complete programming language, this is actually a reasonable approach. The change along the "x-axis" is the side length times $\cos 120°$. The change along the "y-axis" is the side length times $\sin 120°$. Both **sin** and **cos** are PostScript operators, so after having drawn the first line segment of our triangle we could proceed to draw the second one as `currentpoint 120 sin 6 mul add exch 120 cos 6 mul add exch lineto`.

```
6 0 lineto            %construct second line segment
closepath             %construct final line segment
gsave                 %save the graphics state
stroke                %stroke the constructed triangle
grestore              %restore the graphics state
1 0 1 setrgbcolor     %change color to purple
fill                  %fill the constructed triangle
showpage              %view the result
```

Many readers will be surprised by the resulting drawing, which is displayed in figure 2. For example, why is the line width of the black triangle so large? The reason is that the default line width is one unit (along each axis), but we used the **scale** operator to scale the unit to one inch. (The drawing displays as it would on a sheet of letter paper, the boundary of which appears as a light gray outline.) So the line width is a full inch. But this observation implies another puzzle: the line width is one inch, but the stroked line is only half an inch wide.

Overlaid on the black stroked triangle is our filled triangle. This fills the inside of the path that we constructed. This path is therefore the boundary between the outer and inner triangles in our drawing. The **stroke** operator laid down "ink" an inch wide, extending half an inch on each side of the path. But the **fill** operator painted the entire inside of the path, thereby covering everything that was stroked inside the path. PostScript "ink" is completely opaque, so wherever we paint twice we will see only the results of the last painting operation. (With this information you should be able to predict the effect of switching the stroke and fill operators in our program. Try it!)

3.5 Program Details

In addition to our use of coordinate system transformations, we introduced two new elements in this example: color, and the graphics state.

3.5.1 Color

We have seen that in PostScript path construction is distinct from painting (i.e., stroking or filling). Stroking and filling can be done with any color, including any desired shade of gray. As an analogy, we might think of setting the color as choosing your ink. All colors are opaque, including white, so newly applied "ink" completely covers anything beneath it.

To pick a shade of gray we use the **setgray** operator, which takes a single numerical operand between zero and one. As an analogy, we might say the operand signifies the proportion of white ink, where the remainder is black ink. So `0 setgray` selects "black ink" for all subsequent markings on the page, while `1 setgray` selects "white ink".[16] Numbers between 0 and 1 select various shades of gray. To see the effect of this, insert `0.5 setgray` just before stroking the triangle. When you view the result, you will see gray where previously you saw black.

PostScript offers several ways to work with colors other than gray. Since outside of textbooks color graphics in academic publishing remain relatively rare—rarer than they should be, given current technologies—we restrict ourselves to a brief discussion of **setrgbcolor**, which is the most commonly used operator for occasional drawing. This operator implements the popular red-green-blue (RGB) color model. It requires three operands, each a number between zero and one, which give the "illumination" of the red, green, and blue components. In the present example we give the instruction `1 0 1 setrgbcolor` just before filling the triangle: this fully illuminates the red and blue components, thereby setting the color to purple.[17]

[16] Since our first drawing was all in black, it should be clear that this is the default color for a PostScript interpreter. (The initial color space is DeviceGray.)

[17] If you want to produce complex colors, you can find RGB color charts on the Web. Or you can program your own color chart in PostScript after reading section A.

3.5.2 Graphics State

A bit like the man who discovered that he had been speaking prose all his life, we now acknowledge that we have been manipulating the graphics state all along. The current graphics state holds such things as the current path, the current line width, the current color, the current clipping path, the current font, and the current user space. (Adobe 1999a, section 4.2 offers a full discussion.) Often it is useful to temporarily save at least part of the graphics state for reuse. Most of the time we do this with the **gsave** and **grestore** operators.[18]

In the present example, we wish to both stroke and fill the same path. Both stroke and fill perform and implicit **newpath** operation and thereby destroy the current path. We could of course construct the path twice, once for stroking and once for filling. A second approach proves more convenient. Since the current path is part of the graphics state, we can save the graphics state with the **gsave** operator before stroking the path. After stroking it, we can restore the path with the **grestore** operator, so that it is available for filling.

3.5.3 Line by Line

We will now consider this PostScript program line by line. As always the first line of our program is a special comment, which declares it to be a PostScript program, and the last line is the **showpage** operator.

The second line scales the units so that along each axis one unit is now 72 points (i.e., one inch). We then use the **translate** operator to move the origin of user space to the first vertex of our triangle. (The position of the first vertex is arbitrary: for this example we chose to center the triangle horizontally on a sheet of letter paper and to construct the triangle one inch above the bottom of this sheet.) Now we are ready to construct the path describing our equilateral triangle. We begin path construction by moving to the (new) origin, and we then construct our first 6 inch line segment along the bottom of our triangle. Next we move the origin of user space to the second vertex, in preparation for a rotation of user space. We rotate our axes by 120 degrees, which implies that we can construct the second side of our triangle simply by moving right along the "x-axis", exactly as we constructed the first side. The final line segment is constructed for us automatically when we use the **closepath** operator. That completes the path construction for our triangle.

This is a very simple way to construct a perfect equilateral triangle: much simpler than in many point and click drawing programs. (Furthermore, by analogy, we can with equal ease construct a regular polygon with any number of sides.) Once we have constructed our path, we are ready to stroke or fill it. This time we will do both, but this creates a small problem. The **fill** and **stroke** operators each finish with an implicit **newpath** command, which destroys the path we have constructed. We could **stroke** the path and then reconstruct it from scratch for our **fill** operation, but as we saw in section 3.5.2 there is a better way. The **gsave** operator allows us to save a copy of our constructed path, and the **grestore** command allows us to retrieve this saved copy. So before we stroke our constructed path with the default color (black), we save the current path with the **gsave** operator. After our stroke operation, we use the **grestore** operator to restore that path, which we can then fill. Before we fill it, we change the current color to purple. The **showpage** operator will then show us figure 2.

4 Circles and Rectangles

This section illustrates the drawing of circles and rectangles. PostScript arithmetic operators are used to achieve precise placement.

4.1 Drawing Rectangles

Since we know how to draw straight line segments, we obviously know how to draw rectangles. However PostScript also provides specially optimized operators for the construction of rectangles: the **rectstroke**

[18] That is, we push the entire current graphics state onto a special stack, called the graphics state stack, with the **gsave** operator. We use the **grestore** operator to pop the saved graphics state from the graphics state state when we need it again later.

and **rectfill** operators.[19] These accept as operands a corner (x,y) of the rectangle and the displacements (dx,dy) relative to that corner. The **rectstroke** operator strokes the resulting rectangle, and the **rectfill** operator fills the resulting rectangle.

4.2 Drawing Circles

So far our drawings have comprised collections of line segments. For occasional PostScript drawing, the most important remaining path construction operator is **arc**.[20] The **arc** operator constructs part of a circle.[21] It has five operands: the x and y coordinates of the center of the circle, the radius of the circle, the initial angle (at which to start constructing the arc), and the terminal angle. The angles are measured in degrees counterclockwise from the positive x axis. So, for example, `306 396 10 0 360 arc` will construct an full circle with center at (4.25 inch, 5.5 inch) and radius of 10 points, whereas `306 396 10 0 90 arc` will construct only the first quarter of that circle.[22] If the current point is defined, the **arc** operator also constructs a line segment from the current point to the beginning of the arc.

4.3 Application

Figure 3 is based on Hildenbrand and Kirman (1988, Figure AI.1). A circle is exactly contained by a square and exactly contains a square. Our PostScript program to produce this figure is remarkably compact (and could be made more so).

```
%!
/r 250 def                         %define r=250
/r2 r 2 sqrt div def               %define r2=r/1.414
306 396 translate                  %move origin to page center
1 0 0 setrgbcolor                  %set color to red
3 setlinewidth                     %set thick line width
r r r -2 mul dup rectstroke        %stroke large rectangle
45 rotate                          %rotate
1 setlinewidth                     %set thin line width
r2 r2 r2 -2 mul dup rectstroke     %stroke smaller rectangle
0 setgray                          %set color to black
0 0 r 0 360 arc closepath stroke   %stroke large circle
0 0 5 0 360 arc fill               %fill small circle
showpage
```

We begin with something new: we define two variables, r and $r2$. The first is the radius of the circle and large square, which we have simply assigned an arbitrary value of 250, and the second is the radius of the small square, which is computed from the value of the first variable. Next we translate the origin of user space to the exact center of a sheet of letter paper. We set the current color to red and stroke a large rectangle with a thick line. Then we stroke a smaller rectangle with a thin line. We change the current color to black, and we stroke a large circle. Finally we place a small, filled circle at the origin.

[19]These are LanguageLevel 2 operators. We discuss their simplest usage.

[20]Most PostScript drawing relies on eleven commonly used path construction operators (Adobe 1999a, p.191). We have already explored five path construction operators: **moveto**, **rmoveto**, **lineto**, **rlineto**, and **closepath**. In principle that leaves six to go—all associated with drawing arcs and curves—but **arc** is the most important for occasional drawing. The **arc**, **arcn**, **arct**, and **arcto** operators each add an arc of a circle to the current path. The **curveto** operator adds a section of a cubic Bézier curve to the current path, and **rcurveto** does the same operation with relative coordinate displacements. This paper discusses two of these: **arc** and **curveto**.

[21]More accurately, it is an approximation to a circular arc that is constructed from cubic Bézier segments (Adobe 1999a, p.530). This approximation is extremely accurate. Nevertheless it is a good idea to use the **closepath** operator on a 360° arc to account for approximation in the "perpendicular" extension for the line width at the endpoints of the arc. For example, a full circle drawn with a very thick line width will display a "notch" unless the circular path is closed with the **closepath** operator.

[22]Assuming the default user space, of course.

4.4 Program Details

In addition to the new operators for drawing circles and squares, this example introduces three new elements: the use of variables, some basic mathematical computation, and use of the **setlinewidth** operator.

4.4.1 Variables

PostScript has a slightly unusual syntax for the assignment of a value to a variable name. Suppose we wish to assign the value 250 to the name r. Since PostScript uses a postfix notation, we might guess that instead of writing $r = 250$ we would write something like `r 250 =`. This guess would not be far off: we write `/r 250 def`. (Note the slash!) Now whenever the interpreter encounters the name r it will push the value 250 on the operand stack. We have effectively declared a variable and assigned it a value. The **def** operator defines name-value pairs, which is the PostScript approach to assigning values to variables. Details can be found in section A.4.[23]

Appropriate use of variables allows us to make a structured change the behavior of an entire program by simply assigning a different value to the variable. Variables also make our code easier to read and therefore to debug if necessary. (Nevertheless, in a stack oriented language such as PostScript, it is possible and sometimes desirable to avoid the use of variables.)

4.4.2 Basic Math

Our program uses the **dup** stack manipulation operator, which duplicates the object on the top of the operand stack. We also perform some basic arithmetic computations with the **sqrt**, **mul**, and **div** operators. PostScript handles mathematical operations very naturally. Arithmetic operators have self-evident names, take the expected number of operands, and produce the expected result. The operands are popped from the stack, and the result is pushed on the stack. The arithmetic operators **add**, **sub**, **mul**, and **div** each require two numbers as operands to perform addition, subtraction, multiplication, and division. The mathematical operator **sqrt** requires one positive number as an operand for which it produces the square root. These are all extremely useful for the basic computations that may be required for precise placement. Arithmetical, mathematical, and stack manipulation operators are discussed in more detail in section A.1.1.

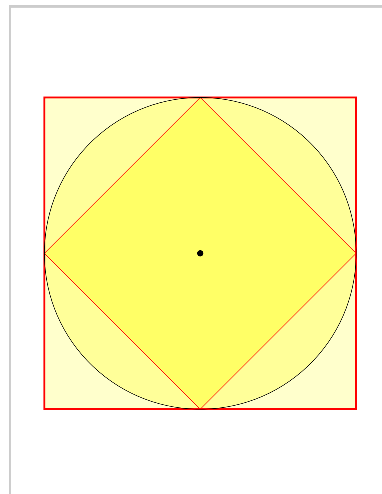

Figure 3: Circles and Squares

4.4.3 Line Width

In section 3 we found that the **scale** operator affected the line width used to stroke a path. We noted that the default line width is one unit, so that when we rescale the axes we also rescale the line width. PostScript also allows direct control of the line width with the **setlinewidth** operator. For example, the instructions `3 setlinewidth` sets the line width to three units.

As an analogy, we might say that a path can be stroked with a "pen nub" of any width we desire. The **setlinewidth** operator does just what it's name suggests: it sets the width of the lines drawn when the path is stroked. This operator takes a single numerical operand: a line width in the current units of the user space. The default width is one unit. Returning to our first program, choose a six point wide "pen nub" by adding `6 setlinewidth` before you stroke the triangle. View the result, and note that half the width of the "ink" falls on each side of the path. Just as when you set a new color, the **setlinewidth** operator affects all subsequent *stroke* commands until you reset it.

[23] Briefly, consider the code **/r 250 def**. The slash push notation allows us to push the literal **name** 'r' onto the operand stack, then the integer 250 is pushed onto the stack, and finally the **def** operator associates the value 250 with the name r in the current **dictionary**.

4.4.4 Line by Line

We will now consider our new PostScript program line by line. As always the first line of our program is a special comment that declares it to be a PostScript program, and the last line is the **showpage** operator.

We begin by defining the variable r to equal 250. (Note again the use of the slash to push the literal **name** r on the stack.) This will be the radius of our large circle and its enclosing square. This number is arbitrary: changes in the value assigned to r can consistently alter the size of the squares and large circle in our drawing by making a single change in our code.[24]

Next we define a second variable, $r2$, which will be the "radius" of the small square. Note that after we push the literal name $r2$ on the operand stack, we use the stack to compute $r/\sqrt{2}$. Finally we use the **def** operator to associate the result of these computations with the name $r2$. The following table illustrates how the operand **stack** behaves during these computations.

/r2	250	2	$\sqrt{2}$	$250/\sqrt{2}$
–	/r2	250	250	/r2
–	–	/r2	/r2	–

The name and value left on the stack are finally popped off by the **def** operator, which associates that pair in the current dictionary.

Next we get ready for our path constructions by translating the origin of user space to the exact center of a sheet of letter paper. We will begin by drawing a large red square with a thick line width, so we change the current color to red and the current line width to 3 units. The instructions `1 0 0 setrgbcolor` set the current color to red: this gives full illumination to red and no illumination to green and blue. The instructions `3 setlinewidth` sets the current line width to 3 units.

Recall that the **rectstroke** operator constructs *and* paints a rectangle based on four operands: a pair (x,y) giving the location of one corner of the rectangle, and a pair of displacements (dx,dy) determining the location of the opposite corner of the rectangle. In this example the values are $(x, y) = (250, 250)$ and $(dx, dy) = (-500, -500)$. Before we can use the **rectstroke** operator, we need to push these four values on the stack. Recall that we assigned $r = 250$. So the code `r r` pushes our first pair of values on the operand stack. Next the code `r -2 mul` computes $r \times -2$ and leaves the result of -500 on the operand stack. Since we need this twice, we simply copy that value with the **dup** operator. So the instructions `r r r -2 mul dup` place the numbers 250, 250, -500, -500 on the operand stack. These are the four values we need on the operand stack, so we can now use the **rectstroke** operator. This will construct a 500 point by 500 point square centered on the page and stroke it with a red line 3 points wide.

The small square is rotated 45° relative to the large square, so we use the instructions `45 rotate` to rotate user space in preparation for its construction. We also reset the line width to 1 unit, since only the large rectangle is drawn with the thicker line width. We are ready to draw the small rectangle. Once again we will need four operands for the **rectstroke** operator, but this time we truly need to compute the location of the rectangle. A side of the smaller rectangle must be smaller than a side of the larger rectangle by a factor of $1/\sqrt{2}$. Recall that the code `/r2 r 2 sqrt div def` effectively made the assignment $r2 = r/\sqrt{2}$. We can therefore construct our smaller rectangle just as we did the larger one, except that we use $r2$ instead of r in our construction. Thus the **rectstroke** operator will construct a $500/\sqrt{2}$ point by $500/\sqrt{2}$ point square centered on the page, but rotated by 45°, and then stroke it with a red line 1 point wide.

Next draw the black circle that is inscribed in the large square and in which the small square is inscribed. We change the current color to black with the instructions `0 setgray`. Then we construct and stroke this circle with the instructions `0 0 r 0 360 arc closepath stroke`. Finally we place a small, filled circle at the origin, which is the center of the page. (The yellow background fills are left as an exercise.)

5 Adding Text to a Drawing

Until now, we have entirely neglected the topic that is the main focus of most introductions to PostScript: the placement and manipulation of text. Since this paper focuses on PostScript *drawing*, so we will have little

[24] Of course if we want to rescale everything, including the line widths and the small filled circle, we can just use the **scale** operator.

to say about adding text. Indeed, if you need to add more than a minimal amount of text to your drawings, it is probably most efficient to use an application that allows you to import your PostScript drawing and annotate it.

For simple labels, however, you may wish to include the text for your drawings in your PostScript Program. Conceptually this is simple.

- Select your font.
- Place your text at a specified location.
- "Print" the text on your drawing.

5.1 Choose Your Font

In principle you can use any PostScript, TrueType, or OpenType font to add text to your PostScript drawings.[25] For occasional drawing, however, the practical options are a fairly small set: you will want to use fonts that you can be confident are present in every PostScript interpreter that might be used to render your drawing. The original Apple LaserWriter (1985) had 13 built-in fonts: the Times, Helvetica, and Courier font families, plus the Symbol font. Every PostScript interpreter will have access to fonts using these names. Indeed, a superset of 35 fonts can be considered standard: add the Bookman, Palatino, New Century Schoolbook, Avant Garde, and Helvetica Narrow families, along with Zapf Chancery Medium Italic and Zapf Dingbats. Adobe considers 136 fonts to be "core" in Adobe PostScript 3, but unless you know ahead of time which fonts will be available where your PostScript drawing will be interpreted, it is best to stick to the basic set of 35.[26]

The Adobe PostScript fonts typically come in families of four. The four members of the Times family are Times-Roman, Times-Italic, Times-Bold, and Times-BoldItalic. The Helvetica family comprises Helvetica, Helvetica-Oblique, Helvetica-Bold, and Helvetica-BoldOblique. The four members of the Courier family are Courier, Courier-Oblique, Courier-Bold, and Courier-BoldOblique. Some specialty fonts such as Symbol are not part of a family. At four typefaces for each of the eight families plus the three additional fonts we get a standard set of thirty-five fonts.[27]

Once you decide on a font face and size, you are ready to go. Just use the **selectfont** operator to select the font and font-size. For example, `/Helvetica 25 selectfont` sets the current font to Helvetica at 25 points.

5.2 Placing Text

To place text in your drawing, you need to create a string that contains the characters to be painted. A **string** is a set of characters enclosed in matched parentheses: `(this is a string)`.[28] The current point determines the placement of the lower left corner of your text when the string is "printed." Set the current point to where you wish your text to *begin*.

[25] As long as you have access to a LanguageLevel 3 interpreter.

[26] However Helvetica is risky if you care about the precise appearance of fonts in your drawings: to avoid licensing fees, Arial is often substituted when Helvetica is requested. (Even Adobe Acrobat does this now. The easiest visual test is to look for Helvetica's "spur" on the capital G.) You can of course ask your interpreter to use any new fonts that you acquire. For example, in Ghostscript you just add the font name and the location of the file with the font information to Ghostscript's Fontmap file.

[27] Here are the rest: AvantGarde-Book, AvantGarde-BookOblique, AvantGarde-Demi, AvantGarde-DemiOblique, Bookman-Demi, Bookman-DemiItalic, Bookman-Light, Bookman-LightItalic, Helvetica-Narrow, Helvetica-Narrow-Oblique, Helvetica-Narrow-Bold, Helvetica-Narrow-BoldOblique, Palatino-Roman, Palatino-Italic, Palatino-Bold, Palatino-BoldItalic, NewCenturySchlbk-Roman, NewCenturySchlbk-Italic, NewCenturySchlbk-Bold, NewCenturySchlbk-BoldItalic, ZapfChancery-MediumItalic, ZapfDingbats. (These are all documented on the Adobe website.)

[28] A string can also contain matched parentheses, but unbalance parentheses must be escaped: \(or \). Backslashes must also be escaped in strings: \\.

5.3 “Printing” Text

To print your string, use the **show** operator. The **show** operator takes one operand, a string. The **show** operator also requires that the current point and current font be defined. It prints the string in the current font, with the lower left corner at the current point. After the text has been printed, the current point is at the lower right of the string. As an example, to print the text “Print this.” in 40 point Helvetica starting at the center of a sheet of letter paper we could use the code

```
%!
/Helvetica 40 selectfont     %select your font
306 396 moveto               %move to center of page
(Print this.) show           %print ‘‘Print this.’’ on the page
showpage                     %view your work
```

5.3.1 Aligning Text

The example above drives home the point that only left alignment of text is automatic in PostScript: the **show** operator begins painting text at the current point. To center text at a position, we need to shift the current point to the left of that position by half the width that text will occupy. We can obtain that width with the **stringwidth** operator, and we can produce the necessary relative movement with the **rmoveto** operator. As we saw in section 2.3.1, the **rmoveto** operator takes as operands the horizontal and vertical displacements (dx,dy) relative to the current point.

For centered text in the above example, you can precede the **show** operator with

```
dup stringwidth pop -2 div 0 rmoveto
```

This gets the string width, divides it in half, and moves the current point left by the resulting amount. (Figure 4 uses this code to center single characters (the numbers) inside a polygon.) Right justification to the current point is achieved similarly:

```
dup stringwidth pop neg 0 rmoveto
```

In both cases, the use of the **stringwidth** is supplemented by two additional operators that you may not have expected: the stack manipulation operators **dup** and **pop**. These are needed because **stringwidth** consumes its operand and leaves an extra item on the operator stack. The **dup** operator duplicates the string for use by the the **stringwidth** operator, and the **pop** operator pops the superfluous item from the operand stack.

5.4 The Symbol Font

The standard PostScript font set does not support serious mathematics. Of course one can purchase specialized font sets, but even then it is best to rely on an application (such as LaTeX) for the typesetting of complex mathematics.

Occasionally however it is useful to place a few mathematical symbols or Greek letters on a drawing, and the standard Symbol font can often be used for this. The character names and character codes of this font are documented in Adobe Systems Incorporated (1999a, appendix E.12). For occasional use, it is helpful that the characters can be accessed with the **glyphshow** operator, which allows you to work directly with the character name (in the current font). For example, the following code places the text $\sum\alpha$ on the page.

```
%!
/Symbol 40 selectfont        %select Symbol font at 40 points
306 396 moveto               %move to middle of page
/summation glyphshow         %print a summation sign on the page
/alpha glyphshow             %print a Greek alpha on the page
showpage                     %view your work
```

We do not have to use **glyphshow**: we can use the character codes in a string with the **show** operator to achieve the same effect. For example, α has character code 141 and $\sum$ has character code 345 in the Symbol font, so the more compact `(\345\141)show` would also have painted the text $\sum\alpha$ on the page. However for occasional drawing using the Symbol font, it is wiser to use the character names, as these aid in understanding and debugging your PostScript programs.

6 Clipping and Pattern Fills

In sections 2, 3, and 4, we created simple drawings in two steps:

- Construct a path.
- Paint the path (i.e., stroke or fill it).

That is the simple essence of PostScript drawing. This section provides some additional details about these two steps, with an emphasis on the utility of dashed lines.

6.1 Dashed Lines

Economists use dashed lines in their drawing almost as often as solid lines. Fortunately, PostScript makes painting a dashed line as simple as painting a solid line along any path. The key to this is the **setdash** operator.

The **setdash** operator takes two operands: an **array** of numbers that determines the dash pattern, and a number that determines the offset for this pattern. The **array** is just a list of numbers surround by brackets. The numbers determine the dash lengths and gap lengths along a stroked path. For example the array [**5**] indicates a dash of 5 units followed by a gap of 5 units. (Note that the stroke operator "cycles" through the array.) As another example the array [**5 1**] indicates a dash of 5 units followed by a gap of 1 unit. There is no restriction the length of this array, so very complex dash patterns can be specified.

The dash pattern is repeated along the entire stroked path. The **offset** operand determines how far "into" the dash pattern we are at the beginning of the stroked path. For example, after `[5] 5 setdash`, a **stroke** operation would begin with a gap of 5 units. whereas after `[5] 2 setdash`, a **stroke** operation would begin with a dash of 3 units.

In section 3 we learned how to construct equilateral triangle and, by implication, regular polygons with any number of sides. In figure 4 we use that knowledge to construct a six pentagons, which we stroke or fill in various ways. In every case we make use of dash patterns.

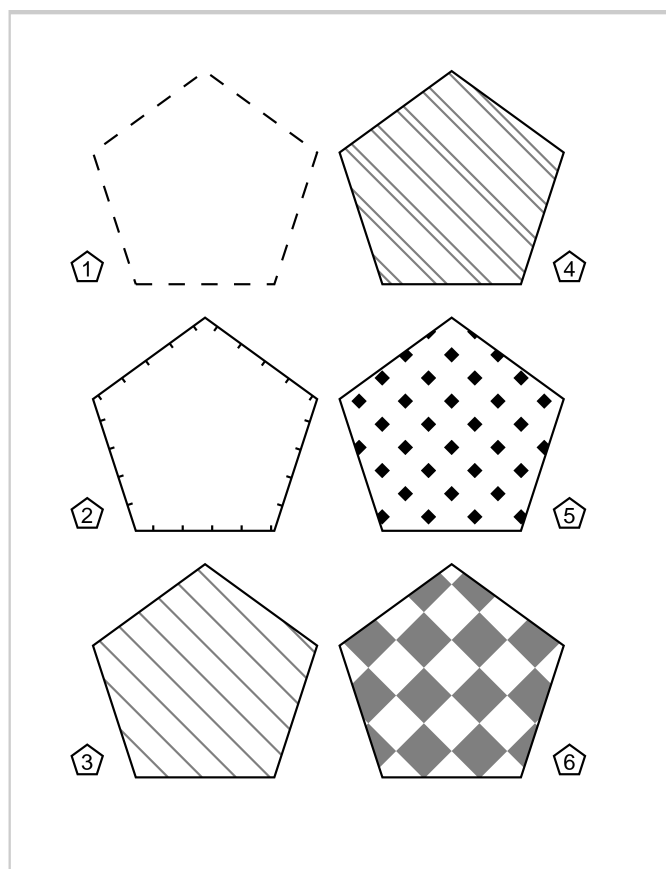

Figure 4: Dashing Patterns

The simplest case is pentagon 1. Any path we can construct we can also stroke with a dashed line. We illustrate this by stroking pentagon 1 with a line width of 2 after the instruction `[15] 0 setdash`. This dash

pattern starts with a dash of 15 units, followed by a gap of 15 units. That initial pattern is then repeated around the entire pentagon.

6.2 Clipping Path

What happens if you **stroke** or **fill** a path that you have constructed in such a way that it goes beyond the boundaries of the page? The answer probably seems very natural: only the part of your drawing that is within the page boundary is painted. Anything beyond that boundary is "clipped": it has no effect on the drawing.

Part of the graphics state is the clipping path. While this is initially the page boundary, it can be constrained further by any path you might wish to construct. Simply construct the path and then invoke the **clip** operator. From then on, any painting will take place only within the new clipping path. You can do this as many times as you want, but the operation is one of repeated intersection: the **clip** operator can only reduce the area enclosed by the clipping path. If you need to *temporarily* shrink the clipping path, then you need to save and restore the larger path. Traditionally this is done by saving and restoring the entire graphics state with **gsave** and **grestore**, as discussed in section 3.5.2.[29] Bracketing every change to the clipping path between a **gsave** and a **grestore** is a reasonable practice, which we follow in this section.

Consider pentagon 2 in figure 4. This illustrates our first use of the clip operator. We are going to add "tic marks" to the inside of the pentagon, since this is one common method of highlighting a specific region. After constructing a pentagon shaped path, we proceed as follows.

```
gsave                  %save the graphics state
clip                   %clip to current path
[2 25] 12 setdash      %set dash pattern and offset
10 setlinewidth        %thick width produces "tics"
stroke                 %stroke the "tics"
grestore               %restore the graphics state
stroke                 %stroke path with solid line
```

First we save the graphics state, which includes our current path and clipping path. (Recall that the current path is the pentagon that we constructed.) We clip to this path so that any painting will take place only inside it. We then set a dash pattern: here we somewhat arbitrarily pick a 2 unit dash followed by a 25 unit gap. When we set the thick line width of 10 units and stroke the pentagon, our dashes show up as "tics" on the interior of the pentagon. Note how the dash length effectively appears as the line width of the tics. Note also how the clipping path restricts the painting to the interior: without this clip operator these tic marks would extend equally on both sides of our path. (Try it!) Of course the **stroke** operator issues an implicit **newpath** operation, thereby destroying our constructed pentagon. However since this path was part of the graphics state when we used the **gsave** operator, we can restore it with the **grestore** operator. After restoring our constructed pentagon as the current path, we stroke it. Since the dash pattern and line width are part of the graphics state, this stroke operation uses their original values. The **grestore** operation similarly restores the original clipping path, so we get the full line width when we stroke our pentagon. The result is pentagon 2 in figure 4.

6.3 Pattern Fills

Economists' drawings often contain shaded areas, where the shading follows a variety of conventions. Section 3.5.1 discussed how arbitrary paths can be filled with arbitrary colors, including arbitrary shades of gray, and this is one common approach to shading. This section discusses some alternatives to such color fills.

Color fills are currently the best choice of shading for electronic documents, since the resolution of computer monitors is inadequate to effectively display most pattern fills at standard magnifications. For the moment, economists should generally reserve their use of pattern fills for print media.

[29]However modern PostScript implementations include the **clipsave** and **cliprestore** operators that may be used instead. Use these operators only if you are sure your PostScript files always have access to interpreters that adequately support LanguageLevel 3.

PostScript is capable of tiling the inside of an arbitrary path with an arbitrary pattern. This facility goes far beyond the needs of occasional drawing is beyond the scope of the present paper.[30] However certain simple pattern fills are common in economic illustrations. For example, hatching and cross-hatching are often used by economists to shade regions of a drawing.[31] This section shows how to produce simple pattern fills simply by using very wide dashed lines.

6.3.1 Hatching

For example, consider the hatching that fills pentagon 3 in figure 4. What we learned in section 2 about drawing line segments and in section 6.2 about clipping suggests one straightforward approach to producing such hatching: clip to the pentagon, repeatedly construct thin line segments passing through it, and then stroke the resulting path. While this works fine, a much simpler solution exists. We will still clip to the pentagon, but we then paint a single *very wide* dashed line passing through it. If the current origin is at the center of a constructed pentagon that has a 1.5 inch "radius", then after clipping to the pentagon we can construct this path as follows.

```
newpath                   %start a new path
-108 -108 moveto          %start path at lower left
108 108 lineto            %construct line to upper right
216 setlinewidth          %very wide line width
[2 20] 0 setdash          %set narrow dash with wide gap
0.5 setgray               %set color to medium gray
stroke                    %stroke the line segment
```

Note the use of the **newpath** operator: in contrast to the previous example, we do *not* wish to stroke the path of the pentagon, which is not destroyed when we clip to it. The next few steps work with some very rough approximations in determining **path** location and line width. These approximation are harmless because nothing outside the clipping path will be painted. The underlying idea is this: if we surely fill an encompassing region, then we will surely fill the pentagon. There is no need for this encompassing region to be small since only the inside of our pentagon will be painted: nothing prevents us from letting the encompassing region be the entire page!

In this case we construct an upward sloping line segment right through the center of our pentagon. The line `-108 -108 moveto` moves to the lower left corner of an encompassing square. The line `108 108 lineto` constructs a line segment to the upper right corner of this encompassing square. The line `216 setlinewidth` makes sure that the line is wide enough when stroked to completely encompass our encompassing square. If we were to stroke our constructed line segment at this point, it would completely fill our pentagon with black.

Instead we give the instructions `[2 20] 0 setdash` which sets a dash pattern of 2 unit of dash followed by 20 units of gap. We can now stroke our line segment and it will fill our pentagon with hatching that is perpendicular to our constructed path. Before we do so, however, we change the current color to a medium gray, since gray hatching has more visual appeal than black. The important thing to note is that the apparent line width of the hatching is actually the dash length of 2 units.

It is hard to imagine a simpler way to generate exactly practically any desired hatching pattern.[32] The "line width" of the hatching and its angle can be easily varied in the obvious way: change the dash pattern and the angle of the constructed line segment. Similarly, if cross hatching is required, just construct a second line segment through the pentagon before the stroke operation. In order to stress the ease of such changes, we produce the patterned hatching that fills pentagon 4 by changing a single line of code: the dashing pattern is set by the instructions `[2 20 2 5] 0 setdash`.

[30] For good discussions see Adobe 1999a, section 4.9 or especially McGilton and Campione (1992, ch.11).

[31] For randomly chosen illustrative examples from textbooks and journal articles see Beavis and Dobbs (1990, ch.3), Glaeser and Shleifer (2002), Lucas and Rossi-Hansberg (2002), or Varian (1984, section 1.18).

[32] As a point of comparison, consider how easy this technique makes it to produce the 16 predefined hatching patterns in the NCAR graphics command language. (As well as many others, of course.)

6.3.2 Stippling

Stippling is another widely used pattern fill. Using dash patterns we can produce a variety of stippling effects very simply. The core idea is to fill an area with the stippling color, then cross hatch the area with thick white lines. Black stippling is illustrated in pentagon 5 in figure 4.[33]

Once again, let the current origin be at the center of a pentagon that has a 1.5 inch "radius", After constructing the pentagon and clipping to it, the stippling can be produced with the following code.

```
0 setgray fill          %black filled pentagon
-108 -108 moveto        %start path lower left
108 108 lineto          %construct line to upper right
108 -108 moveto         %start subpath lower right
-108 108 lineto         %construct line to upper left
216 setlinewidth        %set very wide line width
[20 10] 0 setdash       %set dash pattern and offset
1 setgray               %set color to white
stroke                  %stroke the line segments
```

Since this is *very* similar to the previous example, let us focus on the differences. The first line has been changed to `0 setgray fill`. This fills the pentagon with black. Recall from section 3.5.2 that a **fill** operation performs an implicit **newpath** operation, so we can drop that instruction from our code. We next begin path construction just as in the previous example, drawing a line segment with unit slope through the center of the clipped pentagon. Let us call this segment 1. Then we add a new segment to our construction: this is perpendicular to segment 1 and also extends through the clipped pentagon. Let us call this segment 2. As before, we pick a line width that "encompasses" the clipped pentagon. If we were to stroke our path at this point, it would cause no visible changes: we already have a filled black pentagon. However if we first change the current color to white with the instructions `1 setgray`, then stroking our path would produce (because of the large line width) a filled white pentagon. The trick is to change the current color to white *and* change the dash pattern. We use a dash pattern of `[20 10] 0 setdash`, which paints a 20 unit dash followed by a 10 unit gap. When we paint this in white along segment 1, it paints thick white hatching over our black pentagon. When we continue painting along the perpendicular segment 2, we have effectively painted thick white cross hatching over our black pentagon. The visual appearance, however, is the stippling in pentagon 5 in figure 4.

This basic structure again allows the ready production of many different pattern fills. For example, try making just two small changes: fill the pentagon with red instead of black and change the dash pattern to `[1 10] 0 setdash` to get a widely used (and radically different) pattern fill. (Be sure to predict the outcome before looking at it.)

6.3.3 Just for Fun

Color fills, hatching, and stippling should be adequate to the shading needs of most economists. Nevertheless we will briefly explore a simple way to produce the "checkerboard" shading in pentagon 6 in figure 4. As a shading exercise, this is pure entertainment. However it serves the more serious purpose of introducing two final PostScript operators: **strokepath** and especially **eofill**.

The **strokepath** operator replaces the current path with, roughly, the "outline" of the shape that would result from stroking it given the current graphics state. For example, in the default graphics state, applying **strokepath** to a line segment will produce a rectangular path whose width is the line width. When we apply the strokepath operator to a dashed line, we get a path consisting of outlines of the dashes.

The **eofill** operator behaves just like **fill** operator with one exception: it uses the even-odd rule to determine which points are inside the current path. To apply the even-odd rule to a point, draw a ray from that point in any direction and count the number of times you cross the current path. If that number is odd,

[33] Like other graphics decisions in this paper, the very large stipples are chosen to permit display on relatively low resolution computer monitors. If you are viewing this with a monitor that nevertheless cannot display the stippling, try zooming in on the figure.

the rule says that point is inside the path. For example, if we construct a path that is two circles, points in the interior of both circles are not "inside" the path, whereas points interior to only one circle are considered "inside" the path.[34]

We now use the **strokepath** and **eofill** operators together to produce the "checkerboard" shading in pentagon 6 in figure 4. The strategy is very similar to our previous pattern fill, and indeed much of the code is the same. Once again, let the current origin be at the center of a pentagon that has a 1.5 inch "radius", recalling 1.5 inches is 108 points. After constructing the pentagon and clipping to it, the checkerboard can be produced with the following code.

```
newpath
-108 -108 moveto         %start path lower left
108 108 lineto           %construct line to upper right
108 -108 moveto          %start subpath lower right
-108 108 lineto          %construct line to upper left
216 setlinewidth         %set line width very wide
[36] 0 setdash           %set dash pattern to 0.5" bands
strokepath               %set path to outline of bands
0.5 setgray              %set current color to gray
eofill                   %paint points inside a single band
```

As in our hatching examples, we need to begin with a new path. The subsequent path construction is identical to the stippling example: we construct perpendicular diagonals through the center of the pentagon. Once again we pick a path width easily wide enough to "encompass" our pentagon. Along the constructed path, the instructions `[36] 0 setdash` will therefore create a very thick cross-hatching within the pentagon. A single "dash" will be painted as a half inch wide diagonal whose slope will be plus or minus unity (along the second or first line segment). The **strokepath** operator will create the outline of these bands. The important point is that every point in the pentagon will fall within either 0, 1, or 2 of these bands. The **eofill** operator will only paint those points that fall inside a single band. This creates the checkerboard appearance.

Needless to say, many variants are possible. Changes in the dash pattern or the angle at which the line segments pass through the pentagon can produce many pleasing pattern fills. While this serves to illustrate the power and simplicity of PostScript, the **eofill** operator was the most important concept introduced in the present section.

7 Encapsulated PostScript

Up to now, we have focused on the production of single page PostScript drawings. Come publication time, the publisher will usually request that your PostScript drawings be submitted as *Encapsulated* PostScript (EPS) files. Similarly, if you wish to include your drawing in a presentation or in another document that you are producing, you will often need to produce an EPS version of your drawing.[35] For example, all the drawings in this paper are created as EPS files. Fortunately, it is very simple to turn single-page drawings into *Encapsulated PostScript*.

Conversion by hand is easy, as we discuss below, but there is an even easier way: the free ps2eps converter. For example, if you have installed GSView (or its equivalent), just open your single-page PostScript drawing and pick `File/PS_to_EPS` from the menus. This produces an EPS file suitable for inclusion in larger documents.[36]

[34] This makes Venn Diagram illustration of the concept of "symmetric difference" trivial.

[35] Should another graphics format be required, tools to convert from EPS to other graphics formats are widely available. The mostly widely used free conversion tool is probably pstoedit.

[36] It is important to understand that many WYSIWYG applications, such as those in popular office suites, are not yet able to display an inserted EPS file on screen. If you need onscreen display in such an application, you will need to add a "preview" to your EPS file. You add a preview with pstoedit. (For ease of use, GSView allows this by means of its Edit menu.) Use of the TIFF 6 file format for the preview provides color and good portability, although applications that do not support this preview may object to it, since it is included as part of the EPS file. The so-called Interchange (or EPSI format) provides a grayscale

If you open this EPS file in a text editor, it will look much the same as your original PostScript file, aside from a few new comments. Usually only two of the new comments are absolutely necessary. The first of these says that the document is an EPS file: `%!PS-Adobe-3.0 EPSF-3.0`. This special comment should be the first line in your EPS file.

The second required comment provides a bounding box for your drawing. A bounding box specifies the lower-left and upper-right corners of the smallest rectangle that fully encompasses your drawing, so the form of the comment is: `%%BoundingBox: llx lly urx ury`. (Coordinates are of course in points and must be integer valued.)[37] If you have a very simple drawing you can generally compute the bounding box yourself. You can even print the drawing and use a ruler. However you may want to depend on a PostScript to EPS converter to calculate the bounding box for you.

If you want to be very careful, there are two more comments that are "required" in many drawings (Adobe 1999b). Encapsulated PostScript is part of Adobe's Document Structuring Convention (DSC), which is a set of rules for adding special comments to PostScript files. These comments help applications work with the PostScript files. The DSC says that you should specify the LanguageLevel of your drawing and indicate which resources (e.g., fonts) that your drawing needs. In practice, many EPS files lack this information, and most applications will not demand this information. Nevertheless, a brief discussion follows.

Occasionally you may use one of the newer PostScript operators, in which case the DSC says you must include a comment stating the LanguageLevel. For example, **selectfont** is a LanguageLevel 2 operator, as are **rectstroke**, **rectfill**, and **glyphshow**. So if you use any of these you must include the comment `%%LanguageLevel: 2`. Similarly, **clipsave** and **cliprestore** are LanguageLevel 3 operators, so if you use these you must include the comment `%%LanguageLevel: 3`. While there are many other LanguageLevel 2 and LanguageLevel 3 operators, all listed in Adobe 1999a, the six just mentioned are the most likely to be used for occasional drawing. You cannot count on PS to EPS converters to add a LanguageLevel comment. So if you want to provide this comment with one of your drawings, plan on providing it yourself.

For example, suppose you have created a drawing that uses the selectfont operator (which is a LanguageLevel 2 operator) to select the Times-Roman and Helvetica-Oblique fonts. Suppose also that your drawing is fully encompassed by a rectangle that extends from the point (1 inch,2 inch) to the point (3 inch,4 inch), Then to make an EPS file, add to the top of your PostScript file the following comments.

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 72 144 216 288
%%DocumentNeededResources: font Times-Roman
%%+ font Helvetica
%%LanguageLevel: 2
```

Note how we can continue a DSC comment onto multiple lines by using the `%%+` comment.

In sum, converting PostScript to Encapsulated PostScript is pretty easy. The absolutely necessary changes are two: change the special comment on the first line of your program, and add a bounding box. Additionally, you may wish to conform to the DSC by specifying the LanguageLevel and indicating any resource needs (e.g., fonts used). Generally the applications that import your drawing will take steps against any mistakes you can make. So we end up with three crucial rules that an EPS file must follow:[38]

---

preview as a series of EPS comments, so it is fully portable, but unfortunately in practice many applications simply ignore the Interchange preview. An alternative of course is to rely on applications that have good PostScript support.

[37] The restriction to integer values can be found in Adobe Systems Incorporated (1992b, p.39), the main document structuring manual. Non-integer values can additionally be specified with the **%%HiResBoundingBox** comment (Adobe Systems Incorporated, 1999b, p.6).

[38] As you gain experience with PostScript, you should attend to a couple more details. These boil down to efforts to leave the **stack** and the graphics state in the position it was in before the EPS file was interpreted. It is a good idea to make sure your EPS files does not leave anything on the **stack**. It is very important that an EPS file not use global graphics state commands. This proscribes the use of certain PostScript operators, which are listed in Adobe 1999a, Appendix G. We have used none of the proscribed operators in the present paper. Finally, the obsessive should view the DSC and EPSF specifications to discover some additional DSC comments that are encouraged, including the **%%Pages:** and **%%Page:** comments!

Finally, if you are unsure your drawings will be handled by robust applications, you may want to take a couple precautionary steps as you gain experience with PostScript. For example, you might consider using a private **dictionary** for your EPS drawing. To be extra careful, you may even want to initialize the graphics state (e.g., by setting the current linewidth and

- be a single page drawing
- use the special header **%!PS-Adobe-3.0 EPSF-3.0**
- specify a bounding box

Finally, it is worth addressing a point of possible confusion. Some authors recommend that the EPS file should not use the **showpage** operator, since it instructs the printer to eject the current page. This should never be necessary. Inded, you will find that standard PostScript to Encapsulated PostScript conversion routines leave this operator in your EPS file. Adobe is quite explicit that any application importing an EPS file should redefine **showpage** so that the EPS file does not actually eject the page.

8 Conclusion

Economists often need to produce simple yet precise drawings. Sometimes drawings that are difficult or tedious to produce in a "point-and-click" vector drawing application are very simple to code after a brief exposure to the PostScript page description language. This paper provides examples of such drawings. The goal of the paper is to introduce PostScript drawing in a way that emphasizes tools that are useful for producing the precise geometrical relationships occasionally needed in economic research.

Few economists will turn directly to the PostScript page description language for all of their drawing needs: there are many good applications for data driven graphics, functional representations, and even vector drawing. Each of these can be the right tool, depending on the job. PostScript drawing is best seen as a complement to rather than a substitute for such applications. Three core motivations for turning to PostScript for occasional drawings are precision, speed, and fun. A fourth reason is that occasionally it is necessary to tinker with the PostScript produced by another application.

PostScript is an excellent yet easy to learn tool for the production of simple yet precise representations of geometric relationships. For economists, this proves particularly useful in the fields of social choice, public choice, game theory, and mathematical economics. This paper illustrates both the simplicity and the utility of PostScript drawing for economic research. The results are often as visually exciting as they are useful.

color) rather than assuming the importing application will initialize to the PostScript default values. However, most applications that import your EPS file will be careful enough that these extra precautions will not be necessary.

References

Adobe Systems Incorporated (1992a). *Encapsulated PostScript File Format Specification.* San Jose, CA. URL: `http://partners.adobe.com/asn/developer/pdfs/tn/5002.EPSF_Spec.pdf`.

Adobe Systems Incorporated (1992b). *PostScript Language Document Structuring Conventions Specification.* San Jose, CA. URL: `http://partners.adobe.com/asn/developer/pdfs/tn/5001.DSC_Spec.pdf`.

Adobe Systems Incorporated (1999b). *PostScript Language Document Comment Extensions for Page Layout.* San Jose, CA. URL: `http://partners.adobe.com/asn/developer/pdfs/tn/5644.Comment_Ext.pdf`.

Adobe Systems Incorporated (1999a). *PostScript Language Reference Manual* (3rd ed.). Reading, MA. Known as the "red book". URL: `http://partners.adobe.com/asn/developer/PDFS/TN/PLRM.pdf`.

Beavis, Brian and Ian Dobbs (1990). *Optimization and Stability Theory for Economic Analysis.* Cambridge, UK: Cambridge University Press.

Bourke, Paul (1996). "Bézier Curves." `http://www.swin.edu.au/astronomy/pbourke/geometry/bezier/`.

Carter, Michael (2001). *Foundations of Mathematical Economics.* Cambridge, MA: MIT Press.

Deubert, John (2002, February). "Early Name Lookup With //Double-Slash." *Acumen Journal*, 7–12.

Glaeser, Edward L. and Andrei Shleifer (2002, November). "Legal Origins." *Quarterly Journal of Economics* 117(4), 1193–1230.

Hildenbrand, W. and A.P. Kirman (1988). *Equilibrium Analysis: Variations on Themes by Edgeworth and Walras.* Advanced Textbooks in Economics. Amsterdam: Elsevier Science Publishers B.V.

Karni, Edi and Zvi Safra (2002, January). "Individual Sense of Justice: A Utility Representation." *Econometrica* 70(1), 263–84.

Knuth, Donald Ervin (1997). *The Art of Computer Programming: Seminumerical Algorithms* (3rd ed.), Volume 2. Reading, MA: Addison Wesley Longman.

Kunkel, Gerard (1990). *Graphic Design with PostScript.* Glenview, IL: Scott, Foresmand and Company.

Laffont, Jean-Jacques (1988). *Fundamentals of Public Economics* (Revised English-language ed.). Cambridge, MA: MIT Press. Translated by John P. Bonin and Hélène Bonin.

Lamport, Leslie (1994). *LaTeX: A Document Preparation System* (2nd ed.). Reading, MA: Addison-Wesley Publishing Company, Inc.

Lucas, Jr., Robert E. and Estaban Rossi-Hansberg (2002, July). "On the Internal Structure of Cities." *Econometrica* 70(4), 1445–76.

McGilton, Henry and Mary Campione (1992). *PostScript by Example.* Reading, MA: Addison-Wesley Publishing Company. Known as the maroon book.

Moulin, Hervé (2000, May). "Priority Rules and Other Asymmetric Rationing Methods." *Econometrica* 68(3), 643–84.

Smith, Ross (1990). *Learning PostScript: A Visual Approach.* Berkeley, CA: Peachpit Press.

Varian, Hal R. (1984). *Microeconomic Analysis* (2nd ed.). New York: W.W. Norton & Company.

von Stengel, Bernhard, Antoon van den Elsen, and Dolf Talman (2002, March). "Computing Normal Form Perfect Equilibria for Extensive Two-Person Games." *Econometrica* 70(2), 693–715.

A Programming in PostScript

This section presumes a small amount of programming experience. We begin by reviewing the concept of a device-independent, interpreted, page description language.

Although the PostScript programming language is powerful, it is also very simple. Even a novice can easily produce precise, device independent drawings. By “device independent” we mean that the image is described without reference to any specific rendering device, such as a particular printer or monitor. The simplicity, power, and device independence of the PostScript language made it a very popular way to produce page descriptions. For many purposes, it is a *de facto* standard.

PostScript is an interpreted language: we do not compile our programs into independently ‘executable’ files. Since the object of a PostScript program is to produce visual results, this is very natural: immediate visual feedback can be supplied by a PostScript interpreter.

While the PostScript programming language was developed specifically for graphical applications, it is nevertheless a general-purpose programming language. Many language features will be familiar to anyone with even rudimentary programming experience. It includes standard data types, such as integers, strings, and arrays. It includes standard flow control operators, allowing looping and conditional branching. And it allows for the construction of user defined procedures, which can supplement the built in operators.

A.1 Postfix Notation and the Stack

PostScript is a **stack** based language that uses postfix notation. Essentially this means that programming in PostScript will feel familiar if you have ever used a reverse polish notation calculator. The use of postfix notation just means that operators come after operands. For example, our first example program in section 2 included the code `72 72 moveto`: the **moveto** operator requires two operands, and in our program code the operator comes after these operands.

Closely related to the use of postfix notation is another language feature: PostScript is stack-based. In a PostScript program, we “push” objects onto the operand stack before invoking an operator that requires one or more operands. The operand stack is just a last-in-first-out (LIFO) structure—like a stack of dishes. So when the interpreter encountered the code `72 72 moveto` in our first example program, it pushed 72 onto the operand stack, then pushed another 72 onto the operand stack, and then executed the **moveto** operator. The **moveto** operator then used the top two objects on the operand stack (i.e., the two integers).

So a PostScript operator looks to the operand stack to get any operands it needs. It removes from the top of the stack however many objects it requires: we say the operator “pops” the objects off the top of the stack.

Some PostScript operators also produce results which are pushed onto the top of the stack after they are executed. As an example, suppose we wish to divide 108 by 9. We would use the following PostScript code: `108 9 div`. When the PostScript interpreter encounters this code, it will first push the integers 108 and 9 onto the operand stack. Then it will execute the **div** operator, which pops two values off the stack, divides them, and then pushes the result of the division back onto the operand stack. So the new items on the operand stack are successively:

$$\left|\begin{array}{r}108\\-\\-\end{array}\right\|\begin{array}{r}9\\108\\-\end{array}\left\|\begin{array}{r}12\\-\\-\end{array}\right|$$

When describing an operator, it is useful to in dictate what is removed from and added to the operand stack. For this purpose, Adobe uses a convenient convention for the presentation of operators: list the arguments, then the operator, then the results. In the case of the `div` operator, we have:

`num1 num2` **div** `quotient`

where **quotient=num1/num2**. This means that the **div** operator pops two numbers off the operand stack and then pushes on it one number (the quotient). Of course you must make sure you have pushed the two operands onto the operand stack before invoking the **div** operator.

When an operator returns no result, it is presented as returning `---`. For example, recall our **moveto** operator. We can present it as

`num1 num2` **moveto** `---`

Generally operators that return no result have some side-effect. For example they may change the graphics state. The **moveto** operator of course changes the current point, the position of which is part of the graphics state.

A.1.1 Some Useful PostScript Operators

The PostScript language includes many operators, which are detailed in Adobe Systems Incorporated (1999a). Here are a few that are particularly useful to for occasional PostScript drawing.

Arithmetic and mathematical operators have self-evident names, take the expected number of operands, and have the expected number of returns. The arithmetic operators **add**, **sub**, **mul**, **div**, **mod** require two operands and have one return. The arithmetic operators **abs**, **neg**, **ceiling**, **floor**, **round**, and **truncate** require one operand and have one return. The mathematical operators **sqrt**, **exp**, **ln**, **log**, **sin**, **cos**, and **atan** require one operand and have one return. The operator **rand**, which requires no operands, generates a (pseudo) random integer in the range 0 to $2^{31} - 1$.[39]

Relational and boolean operators have fairly standard names, take the expected number of operands, and have the expected number of returns. The operators have a single return that is boolean (specifically, **true** or **false**). The equality and inequality comparison operators **eq**, **ne**, **ge**, **gt**, **le**, and **lt** require two operands, which should be numbers.[40] The logical comparison operators **and**, **or**, and **xor** require two operands, which should be boolean.[41] The logical comparison operator **not** requires a single operand, which should be boolean.

Some stack manipulation is also required in PostScript programming. The following three stack operators are particularly simple and useful: **dup** duplicates the object on the top of the operand stack, **exch** exchanges the top two objects of the operand stack, **pop** removes ("pops") the top object from the operand stack.

One other stack operator is very useful but a bit more complicated. In the words of Adobe Systems Incorporated (1999a), the **roll** operator performs a "circular shift" of objects at the top of the operand stack. The operator needs to know how much of the stack to manipulate, and how big of a "circular shift" to perform. Assume $n > j > 0$. Then we get an upward roll of the first n elements of the stack, and they roll by j elements.

$$\text{object}_{n-1} \ldots \text{ object}_0 \text{ n j } \textbf{roll} \text{ object}_{j-1} \ldots \text{ object}_0 \text{ object}_{n-1} \ldots \text{ object}_j$$

Consider for example the code

```
5 4 3 2 1 0
6 2 roll
```

The first line would push onto the operand stack `5 4 3 2 1 0`, with 0 at the top of the stack. The second line would change this part of the stack to `1 0 5 4 3 2`.

PostScript does not require $n > j > 0$, so we write more generally

$$\text{object}_{n-1} \ldots \text{ object}_0 \text{ n j } \textbf{roll} \text{ object}_{(j-1) \bmod n} \ldots \text{ object}_0 \text{ object}_{n-1} \ldots \text{ object}_{j \bmod n}$$

If $j < 0$, the direction of the shift is reversed. Consider for example the code

```
5 4 3 2 1 0
6 -2 roll
```

[39] If needed, other random number generators can be based on this one. For example, see Knuth (1997).
[40] String comparison is also possible with these operators.
[41] Bitwise integer comparison is also possible with the logical comparison operators.

The first line would push onto the operand stack `5 4 3 2 1 0`, with 0 at the top of the stack. The second line would roll this part of the stack "downward" by 2 elements to produce `3 2 1 0 5 4`.

A.2 Procedures

Suppose we need to compute the geometric average of 2 and 8. The code fragment `2 8 mul sqrt` will leave the answer 4 on the stack. The code fragment `2 8 {mul sqrt}` does something entirely different: it leaves on the stack the integers 2 and 8 and a third object, known as a procedure object. The braces defer the execution of the enclosed operators: the result is an executable array. One way to execute this array, although not usually the most useful one, is to use the **exec** operator to execute the array object. The code fragment `2 8 {mul sqrt} exec` does just what the first one did: it leaves the answer 4 on the stack. We now explore how to create a procedure that we can execute repeatedly.

A.2.1 Named Procedures

In any programming language, we sometimes wish to create a reusable sequence of operations. We can make a procedure reusable by assigning it a name. In PostScript we do this with the **def** operator, which defines an association between a name and an object, in this case our procedure object. For example, the following code associates the name **gav** with our procedure for calculating the geometric average of two numbers and then uses this procedure twice to compute geometric averages.

```
/gav{mul sqrt}def
2 8 gav
4 9 gav
```

We arbitrarily chose the name **gav** to associate with the geometric average procedure. Notice the use of the slash before the name 'gav'. The slash ensures that the literal **name** 'gav' will be pushed onto the operand stack (rather than any value it may already have in the dictionary stack). Once again we place the body of our procedure between braces, but this time we use the **def** operator to associate this procedure with the name **gav**. Once we have defined this association, we can use our procedure at will. The first time we use it we push the number 4 on the stack, then we use it again and push the number 6 on the stack. (See Adobe 1999a, section 3.5.3 for more detail.)

A.3 Flow Control

Procedures are also used in program flow control. For flow control in a program, PostScript includes standard branching and looping constructs.

A.3.1 Branching

The branching operators are **if** and **ifelse**.

The **if** operator requires two operands: a **boolean** and a **procedure**. If the boolean is true, the procedure is executed.

The **ifelse** operator requires three operands, a boolean and two procedures. If the boolean is true, the first procedure is executed. Otherwise the second procedure is executed.

As an example, consider the following code which implements a standard recursive algorithm for computing the factorial of a positive integer. (Reliance on recursion, as in this algorithm, may lead to a large stack during the computation.)

```
/n!{dup 1 gt {dup 1 sub n! mul}if}def
```

The procedure duplicates the top object on the stack. (A more serious implementation would check that this object is a positive integer, which is required for this procedure to work correctly.) Let us call the number on the top of the stack n. The duplicate of n is used to test whether we have a number greater than unity. If we do, the boolean **true** is pushed on the stack. The **if** operator sees the value true, and so it executes

the procedure in the inner braces. This copies the number, subtracts one from it, computes $(n-1)!$, and multiplies n times $(n-1)!$. Note how reliance on the stack means that we used no variables in this procedure.

A.3.2 Looping

The basic looping operators are **repeat** and **for**. (Two others are *forall* and *loop*.)

The simplest of these is **repeat**, which requires two operands: a positive integer and a procedure. As a simple example, `5{(Hello!) show}repeat` will print “Hello!” five times (assuming a current point is defined).

The **for** loop takes four arguments. In addition to an initial value, an increment, and a limit value, it also requires a procedure to execute at each iteration.

initial increment limit proc **for** —

As a simple example, `1 1 5{(Hello!) show}for` will also print “Hello!” five times. The difference from the previous example is that this would leave five numbers on the stack.[42] This is an important difference, since generally we do not want to simply accumulate values on the stack in this fashion. Instead, the **for** loop should be used when you plan to consume those values. For example, if you want to compute 5!, you could use the commands `1 1 1 5 {mul}for`, which leaves the result of 120 on the stack (but nothing else). This suggests an alternative procedure for computing the factorial of a positive integer.

```
/n! {1 1 1 4 -1 roll {mul}for} def
```

Once again judicious use of the stack renders superfluous any use of variables.

A.4 Variables

As our flow control examples indicate, stack based languages often render needless the use of variables. When desired, we can nevertheless create “variables” by using the **def** operator to associate a name with a value. Say that we wanted to define the **name** x to have a value of 5. The PostScript to do this is: `/x 5 def`. Notice the use of the slash before the **name** x. The slash ensures that the literal **name** x will be pushed onto the operand stack (rather than any value it may already have in the dictionary stack). Then 5 is pushed onto the stack. Then the **def** operator associates the value 5 with the name x in the current **dictionary**, either as a new entry (if x is not in the dictionary) or as a new value for an existing name (if x is already in the current **dictionary**).

A.4.1 Scope

Sometimes we want to use a name without overwriting any pre-existing value. For example, we may want a variable name to be local to a procedure. We can use a dictionary to limit the variable scope in this fashion. A dictionary of names and associated values can be put on the operator stack by bracketing these pairs with `<<` and `>>`.[43] This dictionary can then be pushed on the dictionary stack with the begin operator and popped from the dictionary stack with the end operator.

As an example, we can rewrite our factorial procedure in a more traditional way, so that it uses a variable.

```
/n!{
<< /nfac 1 >>
begin
1 1 3 -1 roll {nfac mul /nfac exch def}for
nfac
end}def
```

[42] If **str** is a string, you can print these out these five numbers with **5 {str cvs show}repeat**. As an exercise, predict what order they will be in.

[43] The << pushes a mark object on the operand stack, while >> creates a dictionary from the key-value pairs following the nearest mark.

Of course this code looks rather ugly and inefficient after seeing the stack oriented approach above, but it does illustrate how a dictionary can be used by a procedure to limit the scope of a variable. The code has another pedagogical advantage as well: it effectively illustrates the difference between an executable name and a literal name. When the PostScript interpreter encounters the name **nfac**, it executes it. That is, it pushes on the operand stack the value associated with it in the current dictionary. In contrast, when it encounters **/nfac**, it pushes on the operand stack the literal name, to which we then assign a new value using the **def** operator.

A.5 Application: Drawing Arrows

We now put a few of these programming concepts to work to create a useful procedure. Economists often need to draw an arrow just as we can draw a line in PostScript: from the current point to a specified endpoint. PostScript does not offer a built in operator for this. The most obvious reason is that the problem of drawing such an arrow is not well specified: there are as many different ideas of what constitutes a nice arrowhead as there are people who have thought about it. Additionally, there is the problem of scaling. An arrowhead that looks nice on a 10 point line may look quite wrong on a 1 point line, despite being appropriately scaled. As can be seen in figure 5, the following procedure definition produces arrowheads that look good over a reasonable range of line widths. It has also been written to allow easy modification to reflect the tastes of the user. Specifically the relative width of the arrowhead can be set to another value—e.g., a larger value to go with small line widths—or computed based on the current line width. This is a pedagogical implementation, so it has limitations (e.g., in the handling of dashed lines), but it is still very useful.

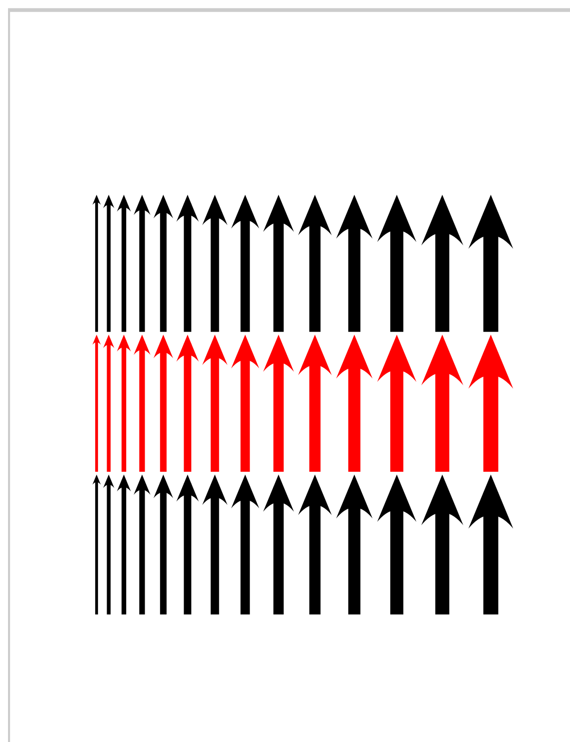

Figure 5: Arrows

A.5.1 Curves

For user convenience PostScript provides the **arc** operator, but an arc is actually drawn as an extremely close Bézier curve approximation to the true arc. PostScript also allows direct drawing with Bézier curves by means of the **curveto** operator.[44]

The **curveto** operator appends a section of a cubic Bézier curve to the current path. A cubic Bézier curve has four control points, and the current point is taken to be the first of these. Following Adobe 1999a, we will call this current point (x_0, y_0). The other three control points are supplied as operands to the **curveto** operator. We will call these (x_1, y_1), (x_2, y_2), and (x_3, y_3). The curve begins at the current point (x_0, y_0) and ends at the end point (x_3, y_3), which then becomes the new current point. The shape of the path between (x_0, y_0) and (x_3, y_3) is controlled by the other two points. Leaving (x_0, y_0), it moves toward (x_1, y_1). Ending at (x_3, y_3), it approaches from the direction of (x_2, y_2). Imagine line segments from (x_0, y_0) to (x_1, y_1) and from (x_2, y_2) to (x_3, y_3). The lengths of these segments might be thought of as the "force" with which the curve is pulled toward the control points (x_1, y_1) and (x_2, y_2). The curve is also tangent to these segments, which is a key property. Another property that can be useful is that the curve is enclosed by the convex hull of the four points. The next section contains a useful application.

[44]Bézier curves are a standard, very flexible way of constructing curves in graphics applications. Adobe 1999a, p.565 has an introductory discussion, and additional accessible detail can be found in Bourke (1996).

A.5.2 Arrow Code

This sections contains the most complex code in this paper, and it has been written to illustrate a few programming possibilities. We will carefully consider a few of the details. The object is to define a procedure **arrowto** that we can use just like we are used to using **lineto**. So we begin by pushing the literal name **arrowto** on the stack, followed by a procedure, followed by the **def** operator. In broadest outline, we have `/arrowto { ... } def`, where the dots indicate the procedure body. Of course the procedure body is where we will be doing all our work. We want the procedure body to construct a line segment from the current point to an endpoint specified by the top two operands on the operand stack.

Take a look at the procedure body. The first thing we do there is start a dictionary by pushing a mark object on the operand stack with the << operator. This dictionary is going to hold all the variable definitions used by our procedure. Put another way, we are going to make sure that all the variables in this procedure are strictly local to the procedure. (It is possible to write this procedure to just use the stack and avoid defining any variables, but the use of variables will be a great aid to understanding the procedure.) Our procedure anticipates the presence of two numbers on the top of the operand stack, which are the coordinates of the end point for our arrow. We will assign these to the variables *tipy* and *tipx* in our dictionary.

Once we have pushed a mark object on the operand stack with the << operator, we push the literal name *tipy* on the stack and roll the y coordinate to the top of the stack. Then we push the literal name *tipx* on the stack and roll the x coordinate to the top of the stack. To see how this works, let us suppose that we are drawing an arrow to the point (72,144) and watch how the stack changes. (This stack illustration leaves out the transient changes involving the operands for the **roll** operators.)

144	<<	/tipy	144	/tipx	72
72	144	<<	/tipy	144	/tipx
–	72	144	<<	/tipy	144
–	–	72	72	<<	/tipy
–	–	–	–	72	<<

The crucial thing to notice is that the mark object can be manipulated on the operand stack like any other object. Thus we can use the **roll** operator to move the endpoint coordinates above the mark, so that each can be used as the value of a name-value pair. The dictionary we will therefore associate the value 72 with the name tipx and the value 144 with the name tipy.

We continue to accumulate name-value pairs on the stack in this fashion. Note that the **currentpoint** operator pushes on the stack two values representing the current point. The **exch** operator is a stack manipulation operator that switches the order of the top two objects on the operand stack. Note that while the mark object << is on the operand stack, we can engage in any calculations we wish to produce the value to be associated with a name. For example, we use the **cos** and **sin** operators to produce the cotangent of 22.5° as the value of *tip* (which is the (relative) extension of the arrowhead beyond the arrow tail) since a total angle of 45° for the arrow tip is visually pleasing. The >> operator will place all the name-value pairs in a dictionary and then pop the mark from the stack. At this point there is an unnamed dictionary object on the top of the operand stack. The **begin** operator pushes this dictionary on the dictionary stack. Since this (unnamed) dictionary is at the top of the dictionary stack, it will hold all new definitions (or redefinitions) until we pop it from the dictionary stack with an **end** operator.

```
/arrowto {                          %tipx tipy  *arrowto*  ---
<<                                  %push mark on stack
/tipy 3 -1 roll                     %arrow tip y coordinate
/tipx 5 -1 roll                     %arrow tip x coordinate
/tailx currentpoint                 %arrow start x coordinate
/taily exch                         %arrow start y coordinate
/tip 22.5 cos 22.5 sin div          %arrow tip: 45 degree angle
/headwidth 3                        %head width is 3 line widths
>>                                  %create dictionary of "local variables"
begin                               %push on dictionary stack
```

```
/dx tipx tailx sub def            %define dx along arrow
/dy tipy taily sub def            %define dy along arrow
/angle dy dx atan def             %arrow angle relaitve to origin
/arrowlength
  dx dx mul dy dy mul add sqrt    %compute length of arrow
def
/tiplength
  tip currentlinewidth 2 div mul  %compute length of arrow tip
def
/base
  arrowlength tiplength sub       %Compute where the arrowhead joins the tail.
def
/headlength
  tip headwidth mul neg           %compute arrowhead length given head width
def
gsave                             %save graphics state
currentpoint translate            %translation to start of arrow
angle rotate                      %rotate user space so arrow points right
base 0 translate                  %translation to end of arrow base
0 0 lineto stroke                 %construct and stroke line
tiplength 0 translate             %translation to tip of arrow
0 0 moveto                        %move to tip of arrow
currentlinewidth 2 div dup scale  %scale to 1/2 linewidth
headlength headwidth lineto       %construct side of arrowhead
headlength tip add 1
headlength tip add -1
headlength headwidth neg curveto  %construct back of arrowhead
closepath                         %construct side of arrowhead
fill                              %fill arrowhead
grestore                          %restore graphics state
tipx tipy moveto                  %move current point to arrow tip
end                               %pop dictionary fr dictionary stack
} def                             %define the arrowto procedure
```

Once all our definitions are in place, we begin construction of the arrow. Construction of our arrow is broken into pieces: first we stroke a line up to the arrowhead, and then we construct the arrowhead. We start construction of our arrowhead from its end point. The arrowhead has straight sides, and we get a nice looking arrowhead by constructing the back with the **curveto** operator. Once we have completed the arrowhead construction, we restore the graphics state and move the current point to the tip of our arrow. We complete our procedure by popping its dictionary off the dictionary stack with the **end** operator.

Useful Terminology

abs `num1` **abs** |`num1`|

Computes |**num1**|, i.e., the absolute value of the operand **num1**.

add `num1 num2` **add** `sum`

Computes **sum** as the sum of the operands **num1** and **num2**.

Array Arrays in PostScript can be literal or executable. Executable arrays are enclosed in braces instead of brackets; they are called **procedures**. Literal arrays are lists of objects enclosed in brackets. For example, `[0 1.5 true /foo]` is a four element array containing the integer 0, the real 1.5, the boolean value true, and the name /foo. You can get any element of the array with the get operator.

`array i` **get** `object`$_i$

It is important to note that, as in many other languages, an array with n elements has indices running from 0 to $n - 1$.

atan `dx dy` **atan** `angle`

Computes **angle** as the angle whose tangent is **dx**/**dy**.

Boolean Boolean objects have one of two values: **true** or **false**.

ceiling `num1` **ceiling** `int`

Returns the least integer that is greater than the operand.

clip Produces a new clipping path by intersecting the current clipping path with the current path.

Important: **clip** does not perform an implicit **newpath** operation. So generally you should follow **clip** with **newpath**.

cliprestore Restores the last saved clipping path. (See **clipsave**.)

A LanguageLevel 3 operator.

clipsave Saves the current clipping path for later restoration. (See **cliprestore**.)

A LanguageLevel 3 operator.

closepath Closes the current subpath by constructing a line segment from the current point to the starting point of the subpath, as determined by the preceding **moveto** operation.

Comment PostScript comments begin with '%'. Subsequent text on the same line will not be interpreted. (Exception: if `%` is part of a string, it does not start a comment.)

Composite Object PostScript categorizes each object as simple, such as a number or a name, or composite, such as an array or a dictionary. The value of a composite object is placed in virtual memory, not on the stack. The stack contains only references to the values of composite objects.

CTM Changes made to the coordinate system of the default user space are summarized in an array known as the current transformation matrix (CTM). See the discussion of user space.

The CTM ensures the proper device rendering of paths painted in user space, regardless of the coordinate transformations applied to user space. See Adobe Systems Incorporated (1999a, section 4.3) for details.

currentpoint --- **currentpoint** `x y`

Pushes on the stack two numbers representing the coordinates of the current point (x, y) in the current user space.

def PostScript associates names with a number of built-in actions: the PostScript operators. Like any modern programming language, it also allows the user to associate a name with a **number** or a **procedure**. The **def** operator is used to define such associations. For example, if the interpreter encounters `/x 25 def`, it will push the literal name **x** on the stack, followed by the number **25**, and then finally the **def** operator will pop this pair off the operand stack and enter them as a *key-value* pair in the current **dictionary**. Similarly, if the interpreter encounters `/average {add 2 div} def`, it will push the literal name **average** on the stack, then the **procedure** object, and finally the **def** operator will pop this pair off the operand stack and enter them as a *key-value* pair in the current **dictionary**.

Dictionary We write our PostScript programs using the names of PostScript operators. We also want to be able to assign names to objects we create—roughly, to be able to use user-defined variables and functions. We do this by associating a name with a number or procedure. Each time we define a variable or procedure, the name and the "value" (as the definition is called) are placed in the current dictionary (which is normally `userdict`). Even the definitions of the core PostScript operators are stored in a dictionary, known as `systemdict`.

There is a dictionary stack, and `userdict` is usually at the top. When the PostScript interpreter encounters a name in your program, it searches the dictionary stack sequentially for the first instance of that name in a dictionary. This means that you can redefine the value of any PostScript operator: if you put your definition in `userdict` it will be found before the PostScript definition and therefore used instead. (As long as your definition is in the current dictionary, it will be found first and therefore used.)

From a programming perspective, dictionaries provide a method for variable scoping. A procedure can create a dictionary to contain the definition of variables local to that procedure.

This paper focuses on a convenient method of dictionary creation introduced in LanguageLevel 2: pairs of literal names and associated values bracketed by << and >>. LanguageLevel 1 uses an apparently more restrictive method of construction but can duplicate this functionality (Adobe Systems Incorporated, 1999a, p.525).

div `num1 num2` **div** `quotient`

Computes **quotient** as the quotient **num1/num2**.

dup `obj1` **dup** `obj1 obj1`

Copies the object on the top of the operand stack and pushes the copy on the operand stack. It is important to note that the value of a composite object is not copied, since this value exists in virtual memory and not on the stack. It is the reference to this value that is on the stack and is copied.

eofill --- **eofill** ---

Behaves just like **fill** with one exception: it uses the even-odd rule to determine which points are inside the current path. To apply the even-odd rule to a point, draw a ray from that point in any direction and count the number of times you cross the current path. If and only if that number is odd, the rule says that point is inside the path. For example, if we construct a path that is two circles, points in the interior of both circles are not "inside" the path.

exec `obj1` **exec** `---`

Execute the top object on the operand stack.

fill Paints the interior of the current path with the current color after implicitly closing any open subpaths. Performs an implicit **newpath** operation (so that the current point becomes undefined).

The *fill* operator uses a winding rule to determine what parts of the page are inside or outside the current path (Adobe 1999a, p.195).

You can create complicated fill patterns if desired (McGilton and Campione, 1992, ch.11).

floor `num1` **floor** `num2`

Computes **num2** as the greatest integer less than the operand **num1**.

for `initial increment limit procedure1` **for** `---`

Repeatedly pushes an incremented value on the operand stack and executes *procedure1*. The **for** operator leaves nothing on the stack *if* the procedure consumes the successive, incremented values, but it does not itself pop these values from the operand stack.

glyphshow `name` **glyphshow** `---`

Starting at the current point, "print" the glyph of the character represented by 'name' in the current font.

A LanguageLevel 2 operator.

grestore Restore the last saved graphics state. More precisely, this operator pops a copy of the last saved graphics state from a special **stack** to which it was saved by the **gsave** operator. This means the path that was current at the time of the **gsave** operation becomes the current path.

gsave Save the current graphics state. More precisely, this operator pushes a copy of the current graphics state onto a special **stack** from which it can be popped by the **grestore** operator. The current graphics state includes the current path.

Graphics State The graphics state holds the current path, the current transformation matrix, the current color, the current linewidth, and various other graphics control parameters. The graphics state can be saved and restored using the **gsave** and **grestore** operators.

if `bool procedure1` **if** `---`

Execute **procdure1** if the boolean object **bool** has value `true`.

ifelse `bool procedure1 procedure2` **ifelse** `---`

Execute **procdure1** if the boolean object **bool** has value `true`. Execute **procdure2** if the boolean object **bool** has value `false`.

LanguageLevel PostScript has added a few operators over time, and operator support of PostScript interpreters is summarized by their LanguageLevel. So far three LanguageLevels have been defined. For example, **selectfont** is a LanguageLevel 2 operator, as are **rectstroke**, **rectfill**, and **glyphshow**. So if you use any of these you must have available a LanguageLevel 2 interpreter. Similarly, **clipsave** and **cliprestore** are LanguageLevel 3 operators, so if you use any of these you must have available a LanguageLevel 3 interpreter. While there are many other LanguageLevel 2 and LanguageLevel 3 operators, all listed in Adobe 1999a, the six just mentioned are the most likely to be used for occasional drawing.

lineto `num1 num2` **lineto** `---`

Construct a line segment from current point to the point **(num1,num2)**.

mod `int1 int2` **mod** `remainder`

Computes **remainder** from the integer division of **int1** by **int2**. (I.e., this is not a true modulo operation.)

mul `num1 num2` **mul** `product`

Computes **product** as the product **num1**$\times$**num2**.

Name PostScript treats as a name object any sequence of regular characters that can not be interpreted as a number. Regular characters are all characters except white space and delimiters.

White space characters are essentially spaces, tabs, and end-of-line markers. Delimiters are the characters `‘(’, ‘)’, ‘<’, ‘>’, ‘[’, ‘]’, ‘{’, ‘}’, ‘/’, ‘%’` .

If a name is preceded by a slash, it is interpreted as a *literal name* object. That is, the PostScript interpreter will simply push the name on the operand stack. Otherwise, it is an executable name. An executable name is treated as a reference to some value in a **dictionary** on the dictionary stack. The interpreter will look for its associated value in the dictionary stack. If the value is a procedure object, the procedure will be executed. If the value is a number, the number will be pushed onto the operand **stack**.

neg `num1` **neg** `−num1`

Computes −**num1** as the additive inverse of **num1**.

newpath Sets the current path to an empty path, so that there is no current point defined.

Most PostScript drawings will comprise several paths, where each new path is initiated by an actual or implicit **newpath** operator. The **stroke** and **fill** operators perform an implicit **newpath**, so they do not need to be followed by a **newpath** operator. However **newpath** is generally desirable after you set a clip path with **clip** or **eoclip**, since these two operators leave the current point unchanged. (However **rectclip** does perform an implicit **newpath** operation.) We also use **newpath** before the arc command to avoid constructing a line from the currentpoint to the beginning of the arc.

Number PostScript distinguishes integers and real numbers. For example, the code fragment `0 0.5` will cause the interpreter to push the integer object 0 on the operand stack and then push the real object 0.5 on the operand stack. Integers and reals can be given various representations (Adobe 1999a, p.28). For example, reals may be either in ordinary decimal form or in scientific notation. Real objects are represented in “single precision” (approximately 8 decimal digits).

Path A path is constructed with path construction operators, such as lineto or curveto. A path is unrestricted in shape. A path can comprise one or more disconnected subpaths.

Path construction does not produce any visible output. A path can be “painted” with the stroke or fill operators to produce visible output. It can also be used to **clip** future painting operations.

The current path is the path under construction. A new path is started by initializing the current path to be empty. This can be done explicitly with **newpath** operator or implicitly by painting the current path. A new path must be initialized by establishing a current point, e.g. with the **moveto** operator.

Point One point is a unit of measurement equal to 1/72 inch. Fractional points are legal. The term 'point' as a unit of measurement is not related to the term 'point' as a dimensionless location.

Procedure A procedure is an executable array. Roughly speaking, we create a procedure by placing a set of PostScript instructions within a pair of braces. The opening brace signals that we do not want immediate execution of the procedure body, which is just a sequence of PostScript instructions. The closing brace produces a procedure object from the procedure body. Often we wish to refer repeatedly to a procedure: the **def** operator allows us to associate a name with a procedure.

A procedure is a composite object.

Since procedures are fundamentally sequences of PostScript operations, they may take operands from the operand stack. Procedures may also "return" values by leaving them on the operand stack. For example, execution of the procedure `{mul sqrt}` takes the top two elements from the operand stack and multiplies them, places the result of the multiplication on the operand stack, and finally takes this number from the operand stack and replaces it with its square root. (Of course a procedure can also be used to change the values of global variables.)

There is a special rule for executable arrays. To quote Adobe 1999a, p.48, "An executable array or packed array encountered directly by the interpreter is treated as data (pushed on the operand stack), but an executable array or packed array encountered indirectly—as a result of executing some other object, such as a name or an operator—is invoked as a procedure." To take an example from the reference manual, if the interpreter encounters `{add 2 div}` it will simply push a procedure object on the operand stack. If it then encounters the **exec** operator, it will execute this procedure. Roughly speaking, this means procedures behave as one expects. For more detail see Adobe 1999a, p.48.

repeat `integer1 procedure1` **repeat** `---`

Executes **procedure1** **integer1** times.

rectfill `x y dx dy` **rectfill** `---`

Paint a filled rectangle. Requires four operands: a corner of the rectangle (x,y) and the displacements (dx,dy) relative to that corner.

A LanguageLevel 2 operator.

rectstroke `x y dx dy` **rectstroke** `---`

Paint a stroked rectangle. Requires four operands: a corner of the rectangle (x,y) and the displacements (dx,dy) relative to that corner.

A LanguageLevel 2 operator.

rotate `num1` **rotate** `---`

Rotate the coordinates of user space by **num1** degrees around the origin.

round `num1` **round** `int2`

Computes **int2** as the integer nearest the operand **num1**.

scale `num1 num2` **scale** `---`

Imposes a "horizontal" scale factor of **num1** and a "vertical" scale factor of **num2**, thereby rescaling the coordinate system of user space without affecting the origin or the orientation of the axes. For example, the point on the page that previously had coordinates $(1, 1)$ will now have coordinates (1/num1,1/num2).

selectfont `/FontName num1` **selectfont** `---`

Sets the current font to **FontName** scaled to **num1** points.

A LanguageLevel 2 operator.

setdash `pattern offset` **setdash** `---`

Here **pattern** is an **array** of numbers that determines the dash pattern: the dash lengths and gap lengths in current unit along a stroked path. For example the array [**5 1**] indicates a dash of 5 units followed by a gap of 1 unit. This dash pattern is then repeated along the entire stroked path. As another example the array [**5 1 5**] indicates a dash of 5 units, followed by a gap of 1 unit, followed by a dash of 5 units. However because it has an odd number of elements this dash pattern is not repeated along the entire stroked path: the stroke operator will "cycle" through the pattern. The array [**5 1 5**] therefore produces the same result as the array [**5 1 5 5 1 5**].

The **offset** operand determines how far "into" the dash pattern we are at the beginning of the stroked path. For example, after `[5 1] 5 setdash` a **stroke** operation would begin with a gap of 1 unit.

setlinewidth `num` **setlinewidth** `---`

Sets the width (perpendicular to the path) of stroked lines to **num** units. A linewidth of 0 is interpreted as a hairline, which is the thinnest line the output device can render.

setrgbcolor `num1 num2 num3` **setrgbcolor** `---`

Sets the color of subsequent paint operations in terms of the red-green-blue color space. Operands should be between 0 and 1, indicating the "illumination" of each color.

Stack PostScript is a stack-based language. A stack is just a last-in-first-out (LIFO) structure—like a stack of dishes. For simple PostScript drawing, the operand stack is most important and has been the focus of our discussions.

However PostScript does have other stacks, and we have been using them. The most important to think about is the dictionary stack. We store the names-value pairs (e.g., named procedures) in the dictionary at the top of the dictionary stack. When desirable, we can always put a new **dictionary** on top of the dictionary stack. For example, the code `<< >> begin` pushes an empty dictionary on the dictionary stack. We can use this for temporary storage (e.g., of variables local to a procedure) and then pop it from the stack with the **end** operator.

String Most often we create a string by enclosing literal text in parentheses. For example, if the interpreter encounters `(Hello)` it will produce a string object and push it on the operand stack. (The string can then be printed in the current font with the **show** operator.) There is also a **string** operator, which takes an integer length as an operand and returns a string of the length with each element initialized to a character code of 0. (If the interpreter encounters `5 string show` it will generally print 5 spaces, but if you want 5 spaces you should explicitly create the string `(     )`, which uses the character code for the space.) For more detail see Adobe 1999a, p.39.

stringwidth `string` **stringwidth** `dx dy`

Returns the change (dx, dy) in the current point location that will occur when **show** is used to paint **string**. (The change dy is zero for Latin character sets.)

stroke — **--- stroke ---**

Paints a line along the current path. Half of the current line width is on each side of the path, the line color is determined by the current color, and **setdash** may be used to make this a dashed line.

The **stroke** operator performs an implicit **newpath** operation (so that the current point becomes undefined).

strokepath — **--- strokepath ---**

Replaces the current path with, roughly, the "outline" of the shape that would result from stroking it given the current graphics state. For example, in the default graphics state, applying **strokepath** to a line segment will produce a rectangular path whose width is the line width.

sub — `num1 num2` **sub** `difference`

The subtraction operator computes **difference** as the **num1−num2**.

systemdict — The read-only dictionary defining the PostScript language operators.

translate — `num1 num2` **translate** ---

Move the origin to a point **(num1,num2)** in the current user space. The two operands determine the translation along the horizontal and vertical axes of the current user space. The current point is not moved.

truncate — `num1` **truncate** `num2`

Computes **num2** as the integer part of the operand **num1**.

User Space — PostScript uses a Cartesian coordinate system for placing objects on the "page". By default, the origin is at the bottom left-hand corner of the page, with standard horizontal and vertical axes. By default, the units along the axes are measured in "points" (where 1 point is 1/72 inch). This is the *default user space.*

The programmer can change the coordinate system of user space through various transformations. For example, the origin can be translated, the axes can be rotated, and the units of measurement along the axes can be (independently) scaled. (See section 3.) This can be done with the coordinate system transformation operators: **translate**, **rotate**, and **scale**. When a programmer makes such changes, the current user space differs from the default user space.

When a path is constructed, it is constructed in the current user space. Changes made to the default user space are summarized in an array known as the current transformation matrix.

Other Useful Stuff

PostScript Error Messages

If your programming gets complex, you are likely to commit errors. In this case the PostScript error messages may be helpful. See Adobe 1999a, sec.4.3.3.

File Input

Occasionally it may prove useful to read in data from another file. The basic procedure is to open a file for reading, read in the data, and then close the file. For example, the following code looks reads in white-space separated numbers that represent points and constructs a path between the points.

```
0 0 moveto
/myfile (test.dat)(r) file def
     {                                  %begin loop
     myfile token                       %get data
          {                             %if true
          myfile token                  %get data
               {lineto}{exit}ifelse
          }
          {exit}                        %if false
          ifelse
     }loop
myfile closefile
```

Efficiency Considerations

Generally efficiency is not a primary consideration in simple PostScript drawing. Nevertheless, here are a couple possible considerations.

Binding Procedures

The elements of a procedure are not evaluated *until* the procedure is invoked. This is called "delayed evaluation". For example, when we defined our procedure `gav` with the code `/gav{mul sqrt}def`, the actual names 'mul' and 'sqrt' are stored in the procedure body: the operations associated with these names are not looked up until we execute the procedure. This is different from an array in which the components *are* evaluated as the array is created. For contrast, the array `[4 9 mul sqrt]` contains one object: the *number* 6.

Consider a small change to our geometric average procedure.

```
/gav{mul sqrt}bind def
```

The bind operator substitutes the actual operators for their names in the procedure, so that there is no need for the PostScript interpreter to look up the names each time we use our procedure. This provides a slight efficiency gain in the form of a modest increase in the speed of execution. It also means that `gav` will always mean the same thing in our program, even if for some reason we later changed the meanings of `mul` or `sqrt`.

More Early Evaluation

Names prepended with a double slash are *immediately evaluated* (Adobe 1999a, sec.3.12.2). If you will often use a procedure that references a constant that is defined in your program, double slash the constant name in the procedure so that it is replaced with its value. This avoids the look-up associated with the constant name.

Double slash can also be used with a procedure name to push the procedure on the stack rather than executing it. So instead of surrounding a single procedure name in braces to provide push a procedure on the stack, you can avoid repetitive look-ups by double slashing the name. Be sure to remember that a procedure that is encountered directly is pushed on the stack, while a procedure that is encountered by executing a name is executed.

A good discussion of these issues can be found in Deubert (2002).

Coordinate Transformations as Matrix Multiplication

The effects of the coordinate transformation operators **translate**, **scale**, and **rotate** can be represented as matrix multiplications. See Adobe 1999a, sec.4.3.3.

Playing with Text

A font is essentially a set of drawing instructions: in PostScript, there is no real distinction between text and graphics. For example, the current graphics state applies to any text added to your drawing. This allows very powerful graphical effects with text.

PostScript includes an operator called **charpath**. This operator requires two operands: a string and a boolean. (The latter is usually *true*.). It then constructs a path at the current point that is a trace of the glyphs represented by the string in the current font. This path is like any other: you can stroke, fill, or clip it. Clipping to text can also be used to provide some startling graphical effects for the dramatic presentation of text (Smith, 1990, ch.8). The following example fills a string with red and strokes it with black.

```
%!
/Helvetica 40 selectfont              %select font face and size
306 396 moveto                        %set current point at page center
(Print this string.) true charpath    %produce outline of text
gsave                                 %save graphics state
1 0 0 setrgbcolor fill                %fill outline with red
grestore                              %restore graphics state
stroke                                %stroke outline with black
showpage
```

Just for Fun: Yin Yang

Multiple Paths

Our triangle drawings construct from line segments a single, connected path. We painted our drawing by either stroking the path with the **stroke** operator or filling it with the **fill** operator. The **stroke** and **fill** operators terminate the current path by implicitly performing a **newpath** operation.

A path need not be connected: it can comprise disconnected subpaths, each begun with the **moveto** operator. But since a **stroke** or **fill** operator affects the entire current path, we cannot set different line widths or colors for different parts of a single path.[45] Most PostScript drawings will therefore comprise several paths.

Arcs

Example 1 (Yin Yang Symbol)
Start a new program. Draw the Yin Yang symbol, centered on a sheet of letter paper.[46]

```
%!
clippath
1 0 0 setrgbcolor fill
/rad 216 def                %assign rad=216
/rad2 rad 2 div def         %assign rad2=rad/2
/rad12 rad 12 div def       %assign rad12=rad/12
306 396 translate           %move origin to page center
0 0 rad 0 360 arc           %construct large circle
1 setgray fill              %fill it with white
0 rad2 neg rad2 270 90 arc  %construct small half circle
0 rad2 rad2 270 90 arcn     %construct small half circle
0 0 rad 90 270 arc          %construct large half circle
```

[45] However the graphics state, which includes line width and color, can be saved with the **gsave** operator and later restored with **grestore** operator. This does allow you to repeatedly use part or all of a single path for stroking or filling. Section 3.5.2 and other examples will show you how to use gsave and grestore to control the graphics state.

[46] This example is adapted from Smith (1990, p.5-8).

```
0 rad2 neg rad12 0 360 arc      %construct little circle
0 setgray eofill
0 rad2 rad12 0 360 arc          %construct little circle
fill                            %fill path with black
0 0 rad 0 360 arc               %construct large circle
stroke                          %stroke with black "ink"
showpage
```

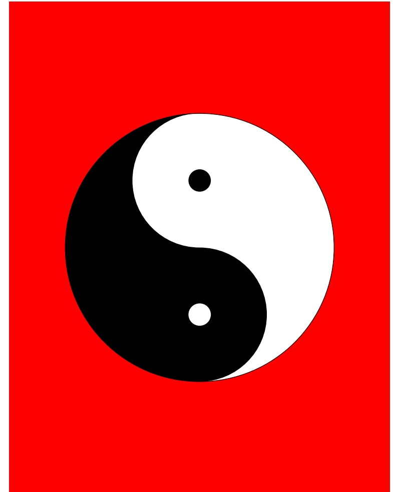

Figure 6: Arcs

The code for this example includes our first example of the assignment of a value to a variable in PostScript. (This is discussed in more detail in section A.4.) We choose the name *rad* for the radius of our symbol. The PostScript code `/rad 216 def` uses the **def** operator to associate the number 216 with this name. The is the PostScript syntax for "assigning" the number 216 to the "variable" *rad*.

The next line assigns half the radius to the variable *rad2*. The PostScript code `rad 2 div` uses PostScripts division operator (**div**) to divide the value of *rad* by 2. One twelfth the figure's radius is similarly assigned to *rad12*. These assignments make the code more flexible: we can easily change the figure size simply by changing a single number (the radius of the figure).

We get ready to draw by translating the origin to the center of a sheet of letter paper. Our first use of the **arc** command constructs a full-radius semi-circle. The arc is constructed from an initial angle of 90 degrees to a terminal angle of 270 degrees, so it describes a semi-circle. This is the first operation in example 1. We then continue this path with a half-radius semi-circle. In each case we provide the coordinates of the center of our half-circle, then we place the remaining three operands on the **stack** and invoke the **arc** operator. The **fill** operator implicitly closes the current path, so we do not have to use the **closepath** operator before filling what we have constructed so far.

All that remains to be done is to draw four circles: a half-size white filled circle at the top, tiny white and black filled circles, and a black stroked circle to border the entire symbol. The construction of the Yin-Yang symbol nicely illustrates how simply we can achieve great precision by drawing with the **arc** operator.

Here are some useful questions to ask about this example.[47] What would happen if we added `180 rotate` right after our translation of the origin? What would happen if we added `0.5 0.5 scale` right after our translation of the origin? How would such rescaling differ from simply changing the radius definition to 108?

[47]Here are the answers. The 180 degree rotation would produce the same symbol except black and white areas would be reversed. (The border would still be black of course.) The rescaling would uniformly shrink the figure size by half. Changing the radius of the figure would have the same effect as the rescaling with one exception: the border thickness would be unchanged at the default value of one point. It is important to understand this exception: scaling will scale the widths of stroked lines (in both the horizontal and vertical dimensions).