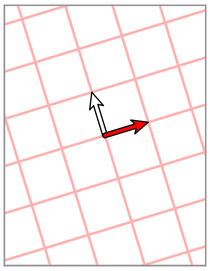
CHAPTER 4

Coordinates and conditionals



We'll take up here a number of drawing problems which require some elementary mathematics and a few new PostScript techniques. These will require that we can interpret absolute location on a page no matter what coordinate changes we have made, and therefore motivate a discussion of coordinate systems in PostScript.

At the end we will have, among other things, a complete set of procedures that will draw an arbitrary line specified by its equation. This is not an extremely difficult problem, but is one of many whose solution will require understanding how PostScript handles coordinate transformations.

4.1. Coordinates

The main purpose of PostScript is to draw something, to render it visible by making marks on a physical device of some kind. Every PostScript interpreter is linked to a physical device—Ghostscript running on your computer is linked to your monitor, and printers capable of turning PostScript code into an image possess an interpreter of their own.

When you write a command like `0 0 moveto` or `1 0 lineto` that takes part in constructing a path, the PostScript interpreter immediately translates the coordinates in the command into coordinates more specifically tied to the physical device, and then adds these coordinates and the command to a list of commands that will be applied to make marks when the path is finally stroked or filled.

Thus a PostScript interpreter needs a way to translate the coordinates you write to those required by the physical device—it has to transform the user's coordinates to the ones relevant to the device, and it must store internally some data necessary for this task.

In fact, PostScript deals internally—at least implicitly—with a total of three coordinate systems.

The first is the **physical** coordinate system. This system is the one naturally adapted to the physical device you are working on. Here, even the location of the origin will depend on the device your pictures are being drawn in. For example, on a computer running a version of the Windows operating system it is apparently always at the lower left. But on a Unix machine it is frequently at the upper left, with the y coordinate reading down. The basic units of length in the physical coordinate system are usually the width and the height of one **pixel** (one horizontal, the other vertical), which is the smallest mark that the physical device can deal with. On your computer screen a pixel is typically $1/75$ of an inch wide and high, while on a high quality laser printer it might be $1/1200$ of an inch on a side. This makes sense, because in the end every drawing merely colors certain pixels on your screen or printer page.

The second is what I call the **page** coordinate system. This is the one you start up with, in which the origin is at the lower left of the page, but the unit of length is one Adobe point—equal to $1/72$ of an inch—in each direction. This might be thought of as a kind of ideal physical device.

The third is the system of **user** coordinates. These are the coordinates you are currently using to draw. When PostScript starts up, page coordinates and user coordinates are the same, but certain operations such as `scale`, `translate`, and `rotate` change the relationship between the two. For example, the sequence `72 72 scale` makes the unit in user coordinates equal to an inch. If we then subsequently perform `4.25 5.5 translate`, the translation takes place in the new user coordinates, so the origin is shifted up and right by several inches.

This is the same as if we had done `306 396 translate` before we scaled to inches, since $306 = 4.25 \cdot 72$ and $396 = 5.5 \cdot 72$.

At all times, PostScript maintains internally the data required to change from user to physical coordinates, and implicitly the data required to change from user to page coordinates as well. The formula used to transform coordinates from one system to another involves six numbers, and looks like this:

$$\begin{aligned} x_{\text{physical}} &= a x_{\text{user}} + c y_{\text{user}} + e \\ y_{\text{physical}} &= b x_{\text{user}} + d y_{\text{user}} + f \end{aligned}$$

PostScript stores these six numbers a, b, etc. in a data structure we shall see more of a bit later.

Coordinate changes like this are called **affine coordinate transformations**. An affine transformation is a combination of a **linear transformation** with a shift of the origin. One good way to write the formula for an affine coordinate transformation is in terms of a matrix:

$$[\, x_\bullet \quad y_\bullet \,] = [\, x \quad y \,] \begin{bmatrix} a & b \\ c & d \end{bmatrix} + [\, e \quad f \,] \ .$$

The 2×2 matrix is called the **linear** component of the coordinate transformation, and the vector added on is called its **translation** component. The translation component records where the origin is transformed to, and the linear component records how relative positions are transformed.

Affine transformations are characterized by the geometric property that they take lines to lines. They also have the stronger property that they take parallel lines to parallel lines. Linear transformations have in addition the property that they take the origin to itself. The following can indeed be rigorously proven:

- *An affine transformation of the plane takes lines to lines and parallel lines tp parallel lines. Conversely, any transformation of the plane with these properties is an affine transformation.*

Later on, we shall see also a third class of transformations of the plane called **projective transformations** (related to perspective viewing). These are not built into PostScript as affine transformations are.

I have said that PostScript uses six numbers to transform user coordinates to physical coordinates, and that these six numbers change if you put in commands like `scale`, `translate`, and `rotate`. It might be useful to track how things go as a program proceeds. In this example, I'll assume that the physical coordinates are the same as page coordinates. When PostScript starts up the user coordinates (x_0, y_0) are the same as page coordinates.

$$\begin{aligned} x_{\text{page}} &= x_0 \\ y_{\text{page}} &= y_0 \ . \end{aligned}$$

or

$$[\, x_{\text{page}} \quad y_{\text{page}} \,] = [\, x_0 \quad y_0 \,] \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \ .$$

If we perform `306 396 translate`, we find ourselves with new coordinates (x_1, y_1). The page coordinates of the new origin are $(306, 396)$. The command sequence `x y moveto` now refers to the point which in page coordinates is $(x + 306, y + 396)$. We thus have

$$[\, x_0 \quad y_0 \,] = [\, x_1 \quad y_1 \,] + [\, 306 \quad 396 \,]$$

or

$$[\, x_{\text{page}} \quad y_{\text{page}} \,] = [\, x_1 \quad y_1 \,] \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + [\, 306 \quad 396 \,] \ .$$

If we now perform `72 72 scale`, we find ourselves with a new coordinate system (x_2, y_2). The origin doesn't change, but the command `1 1 moveto` moves to the point which was $(72, 72)$ a moment ago, and $(72+306, 72+396)$ before that.

$$[\,x_1 \quad y_1\,] = [\,x_2 \quad y_2\,]\begin{bmatrix} 72 & 0 \\ 0 & 72 \end{bmatrix}$$

or

$$[\,x_{\text{page}} \quad y_{\text{page}}\,] = [\,x_2 \quad y_2\,]\begin{bmatrix} 72 & 0 \\ 0 & 72 \end{bmatrix} + [\,306 \quad 396\,]\ .$$

If we now put in `90 rotate` we have new coordinates (x_3, y_3). The command `1 1 moveto` moves to the point which was $(-1, 1)$ a moment ago.

$$[\,x_2 \quad y_2\,] = [\,x_3 \quad y_3\,]\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

or

$$\begin{aligned}[\,x_{\text{page}} \quad y_{\text{page}}\,] &= [\,x_3 \quad y_3\,]\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\begin{bmatrix} 72 & 0 \\ 0 & 72 \end{bmatrix} + [\,306 \quad 396\,] \\ &= [\,x_3 \quad y_3\,]\begin{bmatrix} 0 & 72 \\ -72 & 0 \end{bmatrix} + [\,306 \quad 396\,]\ .\end{aligned}$$

The point is that coordinate changes accumulate. We'll see later (in §5) more about how this works.

4.2. How PostScript stores coordinate transformations

The data determining an affine coordinate change

$$[\,x_\bullet \quad y_\bullet\,] = [\,x \quad y\,]\begin{bmatrix} a & b \\ c & d \end{bmatrix} + [\,e \quad f\,]$$

are stored in PostScript in an **array** `[a b c d e f]` of length six, which it calls a **matrix**. (We'll look at arrays in more detail in the next chapter, and we'll see in a short while why the word 'matrix' is used.) PostScript has several operators which allow you to find out what these arrays are, and to manipulate them.

Command sequence	*Effect*
`matrix currentmatrix`	Puts the current transformation matrix on the stack

There are good reasons why this is a little more complicated than you might expect. The **current transformation matrix** or **CTM** holds data giving the current transformation from user to physical coordinates. Here the command `matrix` puts an array `[1 0 0 1 0 0]` on the stack (the identity transformation), and `currentmatrix` stores the current transformation matrix entries in this array. The way this works might seem a bit strange, but it restricts us from manipulating the CTM too carelessly.

For example, we might try this at the beginning of a program and get

```
matrix currentmatrix ==
[1.33333 0 0 1.33333 0 0 ]
```

The difference between = and ==, which both pop and display the top of the stack, is that the second displays the contents of arrays, which is what we want to do here, while = does not.

The output we get here depends strongly on what kind of machine we are working on. The one here was a laptop running Windows 95. Windows 95 puts a coordinate system in every window with the origin at lower left, with one unit of length equal to the width of a pixel. The origin is thus the same as that of the default PostScript coordinate system, but the unit size might not match. In fact, we can read off from what we see here that on my laptop that one Adobe point is $4/3$ pixels wide.

Exercise 4.1. *What is the screen resolution of this machine in DPI (dots per inch)?*

As we perform various coordinate changes, the CTM will change drastically. But we can always recover what it was at start-up by using the command `defaultmatrix`.

`matrix defaultmatrix` Puts the original transformation matrix on the stack

The **default matrix** holds the transformation from page to physical coordinates. Thus at the start of a PostScript program, the commands `defaultmatrix` and `currentmatrix` will have the same effect.

We can solve the equations

$$[\,x_\bullet \quad y_\bullet\,] = [\,x \quad y\,] \begin{bmatrix} a & b \\ c & d \end{bmatrix} + [\,e \quad f\,]$$

for x and y in terms of $x_\bullet$ and $y_\bullet$. The transformation taking $x_\bullet$ and $y_\bullet$ to x and y is the transformation **inverse** to the original. Explicitly, from

$$P_\bullet = PA + v$$

we get

$$P = P_\bullet A^{-1} - vA^{-1}$$

so it is again an affine transformation. PostScript has an operator that calculates it. The composition of two affine transformations is also an affine transformation that PostScript can calculate:

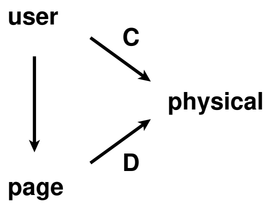
`M matrix invertmatrix` Puts the transformation matrix inverse to M on the stack
`A B matrix concatmatrix` Puts the product AB on the stack

Here, M, A, B, are transformation 'matrices'—arrays of 6 numbers.

Thus, the following procedure returns the 'matrix' corresponding to the transformation from user to page coordinates:

```
/user-to-page-matrix {
  matrix currentmatrix
  matrix defaultmatrix
  matrix invertmatrix
  matrix concatmatrix
} def
```

To see why, let C be the matrix transforming user coordinates to physical coordinates, which we can read off with the command `currentmatrix`. Let D be the default matrix we get at start-up.



The transformation from the current user coordinate system to the original one is therefore the matrix product CD^{-1}: C takes user coordinates to physical ones, and the inverse of D takes these back to page coordinates.

For most purposes, you will not need to use `invertmatrix` or `concatmatrix`. The transformation from user to physical coordinates, and back again, can be carried out explicitly in PostScript with the commands `transform` and `itransform`. At any point in a program the sequence `x y transform` will return the physical coordinates of the point whose user coordinates are (x, y), and the sequence `x y itransform` will return the user coordinates of the point whose physical coordinates are (x, y). If m is a matrix then

```
x y m transform
```

will transform (x, y) by m, and similarly for `x y m itransform`. Thus

```
x y transform matrix defaultmatrix itransform
```

will return the page coordinates of (x, y).

The operators `transform` and `itransform` are somewhat unusual among PostScript operators in that their effect depends on the type of data on the stack when they are used. You, too, can define procedures that behave like this, by using the `type` operator to see what kind of stuff is on the stack before acting.

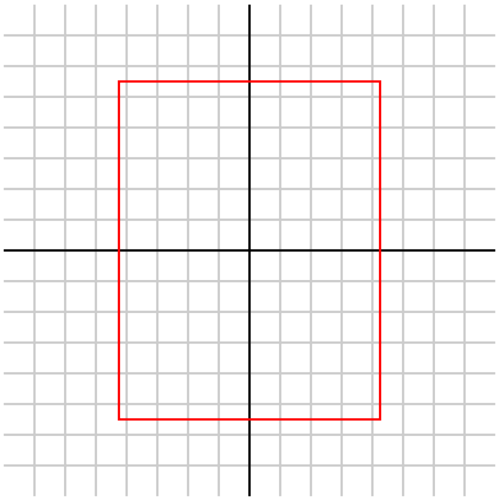
Exercise 4.2. *Write a procedure* `page-to-user` *with two arguments* `x y` *which returns on the stack the user coordinates of the point whose page coordinates are* `x y`*. And also* `user-to-page`*.*

4.3. Picturing the coordinate system

In trying to understand how things work with coordinate changes, it might be helpful to show some pictures of the two coordinate systems, the user's and the page's, in different circumstances. (Recall that the page coordinate system is for a kind of imaginary physical device.)

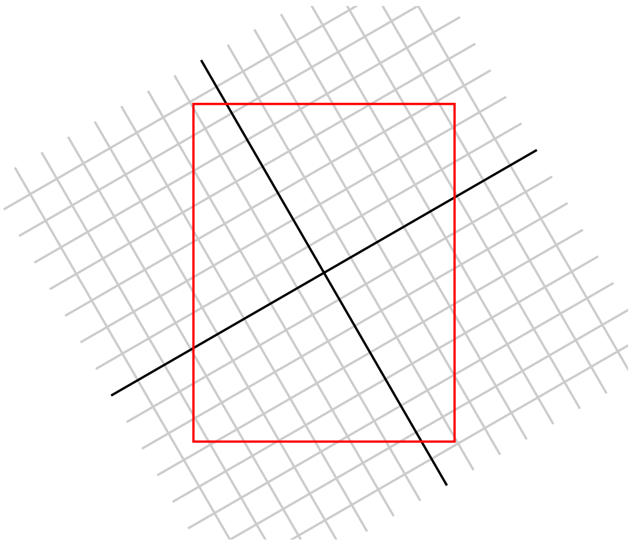
The basic geometric property of an affine transformation is that it takes parallelograms to parallelograms, and so does its inverse. Here are several pictures of how the process works. On the left in each figure is a sequence of commands, and on the right is how the resulting coordinate grid lies over the page.

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



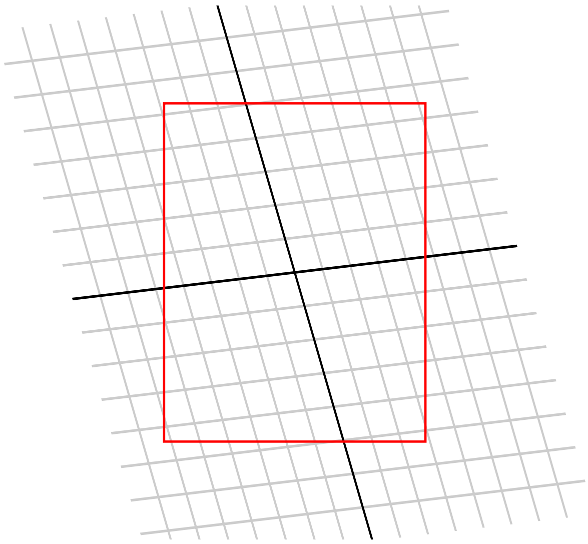
In this figure, the user unit is one inch, and a grid at that spacing is drawn at the right.

```
72 72 scale
4.25 5.5 translate
30 rotate
```



A line drawn in user coordinates is drawn on the page after rotation of 30° relative to what it was drawn as before.

```
72 72 scale
4.25 5.5 translate
30 rotate
0.88 1.16 scale
-18 rotate
```



A combination of rotations and scales can have odd effects after a scale where the x-scale and the y-scale are distinct. This is non-intuitive, but happens because after such a scale rotations take place in that skewed metric.

4.4. Moving into three dimensions

It turns out to be convenient, when working with affine transformations in two dimensions, to relate them to linear transformations in three dimensions.

The basic idea is to associate to each point (x, y) in 2D the point $(x, y, 1)$ in 3D. In other words, we are embedding the two-dimensional (x, y) plane in three dimensions by shifting it up one unit in height. The main point is that the *affine* 2D transformation

$$[\, x_\bullet \quad y_\bullet \,] = [\, x \quad y \,] \begin{bmatrix} a & b \\ c & d \end{bmatrix} + [\, e \quad f \,]$$

can be rewritten in terms of the *linear* 3D transformation

$$[\, x_\bullet \quad y_\bullet \quad 1 \,] = [\, x \quad y \quad 1 \,] \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix} .$$

You should check by explicit calculation to see that this is true. In other words, the special 3×3 matrices of the form

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

are essentially affine transformations in two dimensions, if we identify 2D vectors $[x, y]$ with 3D vectors $[x, y, 1]$ obtained by tacking on 1 as last coordinate. This identifies the usual 2D plane, not with the plane $z = 0$, but with $z = 1$. One advantage of this association is that if we perform two affine transformations successively

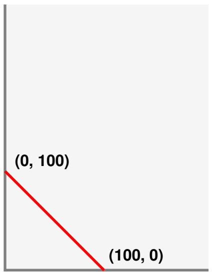
$$[\, x \quad y \,] \longmapsto [\, x_1 \quad y_1 \,] = [\, x \quad y \,] \begin{bmatrix} a & b \\ c & d \end{bmatrix} + [\, e \quad f \,]$$
$$[\, x_1 \quad y_1 \,] \longmapsto [\, x_2 \quad y_2 \,] = [\, x_1 \quad y_1 \,] \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} + [\, e_1 \quad f_1 \,]$$

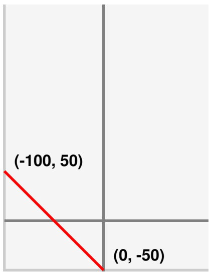
then the composition of the two corresponds to the product of the two associate 3×3 matrices

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix} \begin{bmatrix} a_1 & b_1 & 0 \\ c_1 & d_1 & 0 \\ e_1 & f_1 & 1 \end{bmatrix} .$$

This makes the rule for calculating the composition of affine transformations relatively easy to remember.

There are other advantages to moving 2D points into 3D. A big one involves calculating the effect of coordinate changes on the equations of lines. For example, the line $x+y-100=0$ is visible in the default coordinate system as a line that crosses the screen at lower left from $(0,100)$ to $(100,0)$. If we then `100 50 translate` we will be operating in a new coordinate system where the new origin is at the point that used to be $(100,50)$. The line that was formerly $x+y-100=0$ will have a new equation in the new coordinates.





The line itself is something that doesn't change—in practical terms it's that same collection of pixels at the lower left—but its representation by an equation will change. What is the new equation? We have

$$x_\bullet = x - 100, \quad y_\bullet = y - 50$$

so

$$x + y - 100 = (x_\bullet + 100) + (y_\bullet + 50) - 100 = x_\bullet + y_\bullet - 50 .$$

Now let's look at the general case. The equation of the line

$$Ax + By + C = 0$$

can be expressed purely in terms of matrix multiplication as

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = 0 .$$

This makes it simple to answer the following question:

- *Suppose we perform an affine coordinate change*

$$\begin{bmatrix} x & y \end{bmatrix} \longmapsto \begin{bmatrix} x_\bullet & y_\bullet \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \end{bmatrix} .$$

If the equation of a line in (x,y) coordinates is $Ax + By + C = 0$, what is it in terms of $(x_\bullet, y_\bullet)$ coordinates?

For example, if we choose new coordinates to be the old ones rotated by $90°$, then the old x-axis becomes the new y-axis, and vice-versa.

How to answer the question? The equation we start with is

$$[\,x \quad y \quad 1\,]\begin{bmatrix} A \\ B \\ C \end{bmatrix} = 0\ .$$

We have

$$[\,x_\bullet \quad y_\bullet \quad 1\,] = [\,x \quad y \quad 1\,]\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}, \qquad [\,x \quad y \quad 1\,] = [\,x_\bullet \quad y_\bullet \quad 1\,]\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}^{-1}$$

therefore

$$\begin{aligned} Ax + By + C &= [\,x \quad y \quad 1\,]\begin{bmatrix} A \\ B \\ C \end{bmatrix} \\ &= [\,x_\bullet \quad y_\bullet \quad 1\,]\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}^{-1}\begin{bmatrix} A \\ B \\ C \end{bmatrix} \\ &= [\,x_\bullet \quad y_\bullet \quad 1_\bullet\,]\begin{bmatrix} A_\bullet \\ B_\bullet \\ C_\bullet \end{bmatrix} \\ &= A_\bullet x_\bullet + B_\bullet y_\bullet + C_\bullet \end{aligned}$$

if

$$\begin{bmatrix} A_\bullet \\ B_\bullet \\ C_\bullet \end{bmatrix} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}^{-1}\begin{bmatrix} A \\ B \\ C \end{bmatrix}\ .$$

To summarize:

- *If we change coordinates according to the formula*

$$[\,x_\bullet \quad y_\bullet \quad 1\,] = [\,x \quad y \quad 1\,]\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

then the line $Ax + By + C = 0$ is the same as the line $A_\bullet x_\bullet + B_\bullet y_\bullet + C_\bullet = 0$, where

$$\begin{bmatrix} A_\bullet \\ B_\bullet \\ C_\bullet \end{bmatrix} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}^{-1}\begin{bmatrix} A \\ B \\ C \end{bmatrix}\ .$$

To go with this result, it is useful to know that

$$\begin{bmatrix} A & 0 \\ v & 1 \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & 0 \\ -vA^{-1} & 1 \end{bmatrix}$$

as you can check by multiplying. Here A is a 2×2 matrix and v a row vector. It is also useful to know that

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix}^{-1} = \begin{bmatrix} d/\Delta & -c/\Delta \\ -b/\Delta & a/\Delta \end{bmatrix}, \qquad \Delta = ad - bc\ .$$

Here is a PostScript procedure which has two arguments, a 'matrix' M and and array of three numbers A, B, and C, which returns on the stack the array of three numbers $A_\bullet$, $B_\bullet$, $C_\bullet$ which go in the equation for the transform under M of the line $Ax + By + C = 0$. The procedure starts by removing its components. It begins this with a short command sequence `aload pop` which spills out the array onto the stack, in order. (The operator `aload` puts the array itself on the stack as well, and `pop` gets rid of it.)

```
/transform-line { 1 dict begin
  aload pop
  /C exch def
  /B exch def
  /A exch def
  /M exch def
  /Minv M matrix  invertmatrix def
  [
    A Minv 0 get mul B Minv 1 get mul add
    A Minv 2 get mul B Minv 3 get mul add
    A Minv 4 get mul B Minv 5 get mul add C add
  ]
  end } def
```

This is the first time arrays have been dealt with directly in this book. In order to understand this program, you should know that

(1) the items in a PostScript array are indexed starting with 0;
(2) if A is an array in PostScript, then `A i get` returns the i-th element of A.

Exercise 4.3. *If we set*

$$x_\bullet = x + 3, \quad y_\bullet = y - 2$$

what is the equation in $(x_\bullet, y_\bullet)$ *of the line* $x + y = 1$*?*

Exercise 4.4. *If we set*

$$x_\bullet = -y + 3, \quad y_\bullet = x - 2$$

what is the equation in $(x_\bullet, y_\bullet)$ *of the line* $x + y = 1$*?*

Exercise 4.5. *If we set*

$$x_\bullet = x - y + 1, \quad y_\bullet = x + y - 1$$

what is the equation in $(x_\bullet, y_\bullet)$ *of the line* $x + y = 1$*?*

4.5. How coordinate changes are made

Let's look more closely at how PostScript makes coordinate changes.

Suppose we are working with a coordinate system $(x_\bullet, y_\bullet)$. After a scale change

```
2 3 scale
```

we'll have new coordinates (x, y). What is the relationship between new and old? One unit along the x-axis in the new system spans two in the old, and one along the y-axis spans three of the old. In other words

$$\begin{aligned} x_\bullet &= 2x \\ y_\bullet &= 3y \end{aligned}$$

or

$$[\,x_\bullet \quad y_\bullet\,] = [\,x \quad y\,] \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

or

$$[\,x_\bullet \quad y_\bullet \quad 1\,] = [\,x \quad y \quad 1\,]\begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

If $T_\bullet$ is the original CTM then

$$[\,x_\bullet \quad y_\bullet \quad 1\,]\,T_\bullet = [\,x_{\text{physical}} \quad y_{\text{physical}} \quad 1\,]$$

and for the new coordinates we have

$$[\,x \quad y \quad 1\,]\begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} T_\bullet = [\,x_{\text{physical}} \quad y_{\text{physical}} \quad 1\,]$$

so that the new CTM is

$$T = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} T_\bullet \,.$$

This is the general pattern. To each of the basic coordinate-changing commands in PostScript corresponds a 3×3 matrix, according to the transformation from new coordinates to old ones:

`a b scale`

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`x rotate`

$$\begin{bmatrix} \cos x & \sin x & 0 \\ -\sin x & \cos x & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`a b translate`

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$$

The effect of applying one of these commands is to multiply the current transformation matrix on the *left* by the appropriate matrix.

You can perform such a matrix multiplication explicitly in PostScript. The command sequence

```
[a b c d e f] concat
```

has the effect of multiplying the CTM on the left by

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix} .$$

You will rarely want to do this. Normally a combination of rotations, scales, and translations will suffice. In fact, every affine transformation can be expressed as such a combination, although it is not quite trivial to find it.

Exercise 4.6. *After operations*

```
72 72 scale
4 5 translate
30 rotate
```

what is the user-to-page coordinate transformation matrix?

Exercise 4.7. *Any 2×2 matrix A may be expressed as*

$$A = R_1 \begin{bmatrix} s & \\ & t \end{bmatrix} R_2$$

where R_1 and R_2 are rotation matrices. How to find these factors? Write

$$\begin{aligned} {}^tA\,A &= {}^tR_2\,{}^tS\,{}^tR_1 R_1 S R_2 \\ &= R_2^{-1} S^2 R_2 \end{aligned}$$

since $R^{-1} = {}^tR$ for a rotation matrix R. and ${}^tS = S$. This means that the diagonal entries of S^2 are the eigenvalues of the symmetric matrix ${}^tA\,A$ and that the rows of R_2 are its eigenvectors. In order to find S from S^2 the signs of square roots must be chosen. Describe how to do this, and then how to find R_1. Find this factorization for the shear

$$\begin{bmatrix} 1 & 1 \\ & 1 \end{bmatrix} .$$

Then write a PostScript program that combines rotations and scales to draw a sheared unit circle around the origin. What shape do you get? Why?

4.6. Drawing infinite lines: conditionals in PostScript

We have seen that a line can be described by an equation

$$Ax + By + C = 0 .$$

Recall that the geometrical meaning of the constants A and B is that the direction $[A, B]$ is perpendicular to the line, as long as the coordinate system is an **orthogonal** one, where the x and y units are the same and their axes perpendicular.

The problem we now want to take up is this:

- *We want to make up a procedure with a single argument, an array of three coordinates* `[A B C]`*, whose effect is to draw the part of the line $Ax + By + C = 0$ visible on the page.*

I recall that an **argument** for a PostScript procedure is an item put onto the stack just before the procedure itself is called. I recall also that generally the best way to use procedures in PostScript to make figures is to use them to build paths, not to do any of the actual drawing. Thus the procedure we are to design, which I will call `mkline`, will be used like this

```
newpath
[1 1 1] mkline
stroke
```

if we want to draw the visible part of the line $x + y + 1 = 0$.

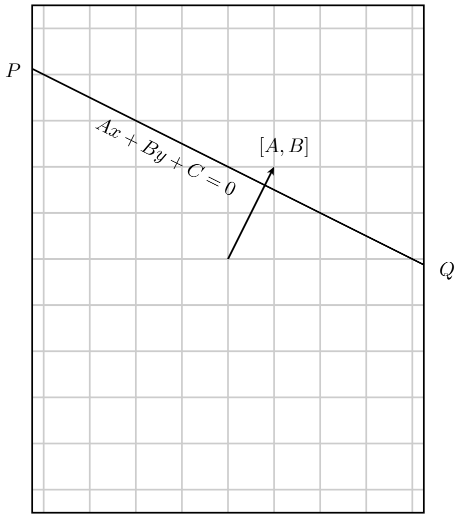
One reason this is not quite a trivial problem is that we are certainly not able to draw the entire infinite line. There is essentially only one way to draw parts of a line in PostScript, and that is to use `moveto` and `lineto` to draw a segment of the line, given two points on it. Therefore, the mathematical problem we are looking at is this: *If we are given A, B, and C, how can we find two points P and Q with the property that the line segment between them contains all the visible part of the line $Ax + By + C = 0$?* We do not have to worry about whether or not the segment PQ coincides exactly with the visible part; PostScript will handle naturally the problem of ignoring the parts that are not visible. Of course the visible part of the line will exit the page usually at two points, and if we want to do a really professional job, we can at least think about the more refined problem of finding them. But we'll postpone this approach for now.

Here is the rough idea of our approach: (1) look first at the case where the coordinates system is the initial page coordinate system; (2) reduce the general case to that one.

Suppose for the moment that we are working with page coordinates, with the origin at lower right and units are points. In these circumstances, we shall divide the problem into two cases: (a) that where the line is 'essentially' horizontal; (b) that where it is 'essentially' vertical. We could in fact divide the cases into truly vertical (where $B = 0$) and the rest, but for technical reasons, having B near 0 is almost as bad as having it actually equal to 0. In this scheme, we shall consider a line essentially horizontal if its slope lies between -1 and 1, and otherwise essentially vertical. In other words, we think of it as essentially horizontal if it is more horizontal than vertical. Recalling that if a line has equation $Ax + By + C = 0$ then the direction $[A, B]$ is perpendicular to that line, we have the criterion:

- *The line $Ax + By + C = 0$ will be considered 'essentially horizontal' if $|A| \leq |B|$, otherwise 'essentially vertical'.*

Recall that our coordinate system is in points. The left hand side of the page is therefore at $x_{\mathrm{left}} = 0$, the right one at $x_{\mathrm{right}} = 72 \cdot 8.5 = 612$. The point is that *an essentially horizontal line is guaranteed to intercept both of the lines $x = x_{\mathrm{left}}$ and $x = x_{\mathrm{right}}$*. Why? Since A and B cannot both be 0 and $|A| \leq |B|$ for a horizontal line, we must have $B \neq 0$ as well. Therefore we can solve to get $y = (-C - Ax)/B$, where we choose x to be in turn x_{left} and x_{right}. In this case, we shall choose for P and Q these intercepts. It may happen that P or Q is not on the edge of the page, and it may even happen that the line segment PQ is totally invisible, but this doesn't matter. What does matter is that the segment PQ is guaranteed to contain all of the visible part of the line.



Similarly, an essentially vertical line must intercept the lines across the top and bottom of the page, and in this case P and Q shall be these intercepts.

So: we must design a procedure in PostScript that does one thing for essentially horizontal lines, another for essentially vertical ones. We need to use a **test** together with a **conditional** in our procedure.

A test is a command sequence in PostScript which returns one of the **boolean values** `true` or `false` on the stack. There are several that we will find useful: `le`, `lt`, `ge`, `gt`, `eq`, `ne` which stand for $\leq$, $<$, $\geq$, $>$, $=$, and $\neq$. They are used backwards, of course. For example, the command sequence

```
a b lt
```

will put `true` on the stack if $a < b$, otherwise false.

Here is a sample from a Ghostscript session:

```
1 2 gt =
false
2 1 gt =
true
```

A **conditional** is a command sequence that does one thing in some circumstances, something else in others. The most commonly used form of a conditional is this:

```
boolean
{ ... }
{ ... }
ifelse
```

That is to say, we include a few commands to perform a test of some kind, following the test with two procedures and the command `ifelse`. If the result of the test is `true`, the first procedure is performed, otherwise the second. Recall that a procedure in PostScript is any sequence of commands, entered on the stack surrounded by `{` and `}`.

A slightly simpler form is also possible:

```
boolean
{ ... }
if
```

This performs the procedure if the boolean is true, and otherwise does nothing.

To apply `if` or `ifelse`, normally you apply a test of some kind. You can combine tests with `and`, `not`, `or`.

We now have just about everything we need to write the procedure `mkline`. Actually, for reasons that will become clear in a moment we make up a procedure called `segment-page` that instead of building the line returns (i.e. leaves on the stack) the two endpoints P and Q, as a pair of arrays of two numbers. We need to recall that `x abs` returns the absolute value of `x`.

```
 % [A B C] on stack
/segment-page { 1 dict begin
aload pop
/C exch def
/B exch def
/A exch def

A abs B abs le
{
  /xleft 0 def
  /xright 612 def
  /yleft C A xleft mul add B div neg def
   % y = -C - Ax / B
  /yright ...  def
  [xleft yleft]
  [xright yright]
}{
  ...
} ifelse
end } def
```

I have left a few blank spots—on purpose.

Exercise 4.8. *Fill in the . . . to get a working procedure. Demonstrate it with a few samples.*

Exercise 4.9. *Show how to use this procedure in a program that draws the line* $113x + 141y - 300$ *in page coordinates.*

Now we cease to assume that we are dealing with page coordinates. We would like to make up a similar procedure that works no matter what the user coordinates are. So we are looking for a procedure with a single array argument `[A B C]` that builds in the current coordinate system, no matter what it may be, a line segment including all of the line that's visible on a page. This takes place in three stages: (1) We find the equation of the line in page coordinates; (2) we apply `segment-page` to find points containing the segment we want to draw, in page coordinates; (3) we transform these points back into user coordinates.

Here is the final routine we want:

```
 % [A B C] on stack
/mkline { 1 dict begin
  aload pop
  /C exch def
  /B exch def
  /A exch def
   % T = page to user matrix
  /T
    matrix defaultmatrix
    matrix currentmatrix
    matrix invertmatrix
    matrix concatmatrix
  def
   % get line in page coordinates
  [
    A T 0 get mul
    B T 1 get mul add

    A T 2 get mul
    B T 3 get mul add

    T 4 get A mul
    T 5 get B mul add
    C add
  ]
   % find P, Q
  segment-page
   % build the line
  aload pop T transform moveto
  aload pop T transform lineto
end } def
```

Exercise 4.10. *Finish the unfinished procedures you need, and assemble all the pieces into a pair of procedures that will include this main procedure. Exhibit some examples of how things work.*

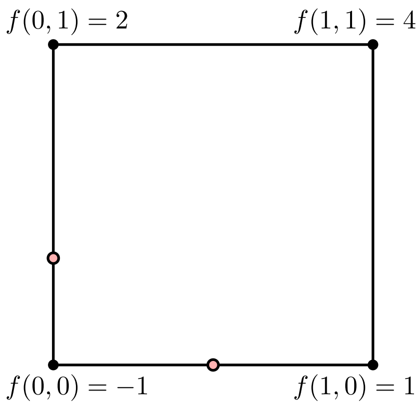
4.7. Another way to draw lines

There is another way to solve the problem posed in this chapter, and that is to find exactly where the line enters and exits the page. This reduces to the following problem: *Suppose* (x_0, y_0) *and* (x_1, y_1) *are the lower left and upper right corners of a rectangle, and* $Ax + By + C = 0$ *the equation of a line. How can you find the points where the line intersects the boundary of the rectangle?*

There are many possibilities, of course—the line could intersect in 0, 1, or 2 points, and in the course of solving the problem you must decide which. We'll deal with a related problem in a more organized fashion later on when we look at the Hodgman-Sutherland algorithm, but the problem here, although somewhat related, is simpler.

We want to define a procedure with three arguments—(x_0, y_0), (x_1, y_1) (corners of the rectangle) and $[A, B, C]$—that returns an array of 0, 1, or 2 points of intersection. Details will be left as an exercise, but I want to explain here some of the features of the procedure that will be applicable in other situations, too.

Let's look at an explicit example to see how things are going to go. Suppose we ask whether the line $f(x, y) = 2x+3y-1 = 0$ intersects the frame of the unit square, and if so where it intersects. The thing to keep in mind here is that the computer is, of course, blind. It can't see anything, so how is it going to figure out where intersections occur? The fundamental criterion is this: *if* P *and* Q *are two points on either side of the line, then* $f(P)$ *and* $f(Q)$ *will have different signs.* In other words, we evaluate $f(x, y)$ at each of the corners of the square, and if we find sides where the signs at the ends are different, we have an intersection. Here is what happens in this case:

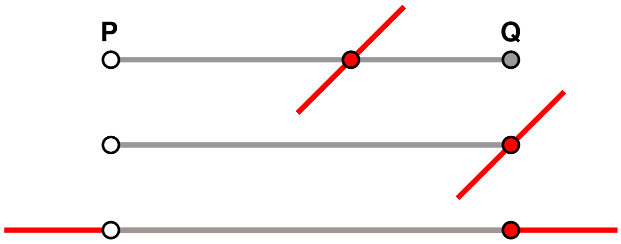


There are two sides where the sign of $f(x)$ is different at the end points. At the bottom the range is -1 to 1, and will be 0 at the midpoint. At the left the range is -1 to 2, which tells us that the intersection point is $1/3$ of the way from bottom to top.

What happens in the procedure? Start at one corner, say the lower left, and then 'walk around' the sides of the rectangle, checking to see if the line $Ax + By + C$ intersects that side. There are a few subtle points to be taken into account.

One subtle thing is to be careful exactly what a **side** is. As we walk around the sides, we traverse them in a particular direction—the sides are **oriented** segments. It turns out to be best to define a side of the rectangle to include its endpoint but exclude its beginning. Thus a side looks like this, metaphorically:

Before I start with details, let's think about the possibilities. (1) There could of course be no point of intersection at all. (2) There could be a single intersection, which could be either be (a) in the middle or (b) at the endpoint. (3) The whole side could be part of the line.



How can we distinguish these cases? Basically, by checking the sign of $Ax + By + C$ at the two ends of the side. Let

$$F_P = Ax_P + By_P + C, \quad F_Q = Ax_Q + By_Q + C \ .$$

Here is a complete logical breakdown:

- If $F_P < 0$ and $F_Q > 0$, there is a single interior point of intersection.
- Same conclusion if $F_P > 0$ and $F_Q < 0$.
- If $F_P < 0$ and $F_Q = 0$ then the endpoint Q is a point of intersection.
- Similarly if $F_P > 0$ and $F_Q = 0$.
- If $F_P = 0$ and $F_Q = 0$ the side is contained in the line.
- In all other cases there is no intersection.

One question is how to deal with the case where the side is contained in the line. It turns out that there is one best thing to do, and that is to treat it exactly the same as other cases where Q is on the line. This will work because on the previous side the point P will be counted, and the procedure will return both P and Q, which is quite reasonable.

The breakdown can be summarized: (1) if $F_P F_Q < 0$ then there is an interior point; (2) if $F_Q = 0$ then Q is the point of intersection; (3) otherwise, there is no intersection.

Another problem is mathematical. In the case of an interior point of intersection, how is it calculated? The function $Ax + By + C$ is equal to $F(P)$ at P, 0 at the intersection point, and $F(Q)$ at Q. The increase of F across the segment is linear, so that

$$F((1-t)P + tQ) = (1-t)F(P) + tF(Q) = F(P) + t(F(Q) - F(P))$$

and if we want to get $F((1-t)P + tQ) = 0$ we set

$$t = \frac{F(P)}{F(P) - F(Q)} \ .$$

I can say some more about what the procedure does. After defining some variables, it puts a `[` down on the stack, then looks at each side in turn. If there is a point of intersection (x, y) it puts it on the stack as an array of two points. Otherwise it does nothing. As the procedure exits it puts `]` on the stack. What's returned on the stack will be thus of the form `[]`, `[[..]]`, or `[[..][..]]`. I can offer one hint, too, for efficiency—start off with P equal to the corner point, and calculate F_P immediately. When you have looked at a side, define the new values of P and F_P to be the current values of Q and F_Q and go on to the next side.

Exercise 4.11. *Define the procedure in detail. Write it so that it will handle any convex closed polygon, that is to say one that bulges out, so that the intersection of a line with it is always either empty, a single point, two points, or a whole side.*

4.8. Clipping

It might be that we don't want to draw all of the visible line, but want to allow some margins on our page. We could modify the procedure very easily to do this, by changing the definitions of `xleft` etc., but this is inelegant, since it would require putting in a new procedure for every different type of margin. There is a more flexible way. There is a third command in the same family as `stroke` and `fill`, called `clip`. It, too, is applied to a path just constructed. Its effect is to restrict drawing to the interior of the path. Thus

```
newpath
72 72 moveto
540 72 lineto
540 720 lineto
72 720 lineto
closepath
clip
```

creates margins of size $1''$ on an $8.5'' \times 11''$ page in page coordinates. If you want to restrict drawing for a while and then abandon the restriction, you can enclose the relevant stuff inside `gsave` and `grestore`. The command `clip` is like `fill` in that it will automatically close a path before clipping to it, but as with `fill` it is not a good habit to rely on this. This is another example of the idea that *programs should reflect concepts*: if what you really have in mind is a closed path, close it yourself. The default closure may not be what you intend.

4.9. Order counts

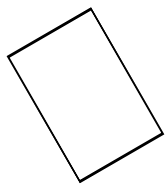
This seems like a good place to recall that the order in which a sequence of coordinate changes takes places is important. Sometimes this is a useful feature, sometimes just a nuisance.

I have said before that non-uniform scaling (i. e. scaling by different factors on the different axes) can have peculiar effects. This is particularly so when applying rotations. It is very important in what order rotating and scaling occur, if the scaling is not uniform.

Here is what happens for each of these sequences:

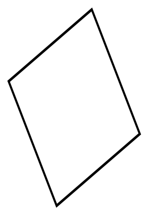
```
30 rotate
1 1.5 scale

newpath
0.5 square
stroke
```



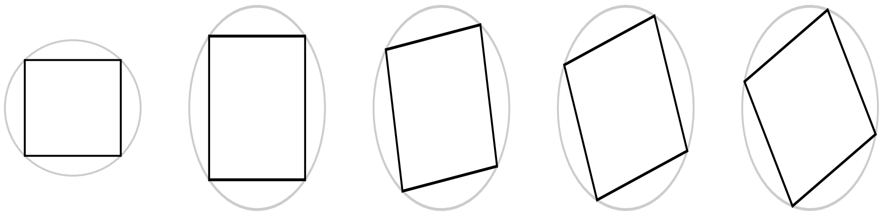
```
1 1.5 scale
30 rotate

newpath
0.5 square
stroke
```



The point to keep in mind is something I have said before—after a coordinate change is made, all further coordinate changes take place *with respect to that new system*. Rotation always preserves the curve $x^2 + y^2 = 1$.

But if x and y are scaled by different factors, a circle will become an ellipse, and rotation takes place *around this ellipse* instead of around a true circle. We can see what happens by doing it in stages:



4.10. Code

The code to draw lines can be found in `lines.inc`.