Mathematics 308 — Geometry

Chapter 4. Drawing lines: conditionals and coordinates in PostScript

We will take up here a number of drawing problems which require some elementary mathematics and a few new PostScript techniques. At the end we will have, more or less, a complete group of procedures that will draw arbitrary lines. AS we shall see, this is not quite trivial to achieve.

1. Drawing infinite lines

In high school you were taught that the equation of a line is of the form

$$y = mx + b$$

where m is the slope of the line and b the y-intercept, the height of the point where the line crosses the y-axis. That is true so far as it goes, but this representation of lines has the fault that not all lines can be written in this way—some lines have infinite slope and no well defined y-intercept. These are the vertical lines, the lines $x = c$ for some constant c. In other words, writing lines in slope-intercept form destroys an intrinsic symmetry between the coordinates x and y. The most versatile way to write lines, preserving this symmetry, is in the form

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

Here we must assume that not both A and B are equal to zero, for if they are then we have no condition at all on x and y. Equivalently, we may assume that

$$A^2 + B^2 \neq 0$$

since $A^2 + B^2 = 0$ if and only if $A = 0$ and $B = 0$. At any rate, the equation for the line can be converted into

$$y = \frac{-C - Ax}{B}$$

as long as $B \neq 0$. If B is equal to 0 then A cannot be 0, and we can write our line as

$$x = \frac{-C}{A} .$$

Recall that the geometrical meaning of the constants A and B is that the direction (A, B) is perpendicular to the line.

The problem we now want to take up is this:

- *We are working in PostScript with a coordinate system whose unit is an inch, and with the origin at the centre of the page. We want to design a procedure with three arguments A, B, C and whose effect is to draw the part of the line $Ax + By + C = 0$ which is visible on the page.*

I recall that **arguments** for a PostScript procedure are items 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 **`make-line`**, will be used like this

```
newpath
1 1 1 make-line
stroke
```

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

The reason this is not quite a trivial problem is that we are certainly not able to draw the entire line. There is only one way to draw linear figures in PostScript, and that is to use **`moveto`** and **`lineto`** to draw a segment of the line, given two points on that line. 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 of the line $Ax + By + C = 0$ which is visible?* 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 at exactly two points, and if we want to do a really professional job, we can at least about the more refined problem of finding them, too. But we will postpone this for now.

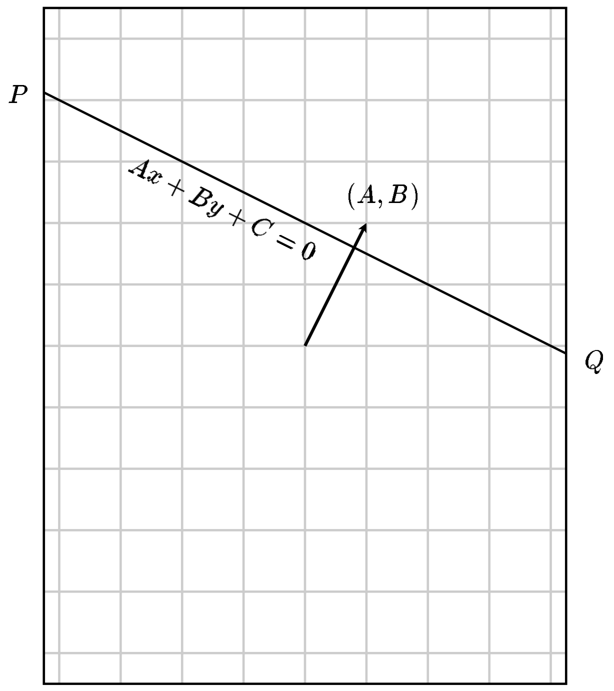
Here is the rough idea of our approach. We will divide the problem into two cases, (1) that where the line is 'essentially' horizontal, and (2) that where it is 'essentially' vertical. We could in fact divide the cases according to what is suggested by our initial discussion—i.e. according to whether or not $B = 0$. But for technical reasons, having B near 0 is almost as bad as having it actually equal to 0. Instead, we want to classify lines as 'almost horizontal' and 'almost vertical'. In this scheme, we shall consider a line almost horizontal if its slope lies between -1 and 1, and otherwise almost vertical. In other words, we think of it as 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 horizontal if $|A| \le |B|$, otherwise vertical.*

Recall that our coordinate system is in inches, centred on the page. The left hand side of the page is therefore at $x_{\text{left}} = -4.25''$, the right one at $x_{\text{right}} = 4.25''$. The point is that *a horizontal line is guaranteed to intercept both of the lines $x = x_{\text{left}}$ and $x = x_{\text{right}}$*. Why? Since A and B cannot both be 0 and $|A| \le |B|$ for a horizontal line, we must have $B \ne 0$ as well. Therefore we can solve for y, given x:

$$y = \frac{-C - Ax}{B} .$$

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, a 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 horizontal lines, another for 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 $\le$, $<$, $\ge$, $>$, $=$, and $\ne$. They are used backwards, of course. The line

```
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, otherwise does nothing.

We now have everything we need to write the procedure, except that we need to know that `x abs` returns the absolute value of `x`.

```
/make-line {
8 dict begin
/C exch def
/B exch def
/A exch def

A abs B abs le
{
/xleft -4.25 def
/xright 4.25 def
/yleft C neg A xleft mul sub B div def
% y = -C - Ax / B
/yright ...  def
xleft yleft moveto
xright yright lineto
}
{
...
} ifelse
end
} def
```

I left a few blank spots—on purpose.

Exercise 1.1. *Fill in the* `...` *to get a working procedure. Demonstrate it with a few samples.*

Exercise 1.2. *Modify the procedure above to one called* `make-line-default` *that works with the bold default coordinate system, the one with the origin at bottom left, unit of length one point.*

2. Margins

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
-3.25 -4.5 moveto
6.5 0 rlineto
0 9 rlineto
-6.5 rlineto
closepath
clip
```

creates margins of size $1''$ on an $8.5'' \times 11''$ page. 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. The point is 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.

3. Coordinates

In the first section of this chapter, we saw how to draw lines when the coordinate system on the page was set up in a fixed way. In this section we shall see how to draw lines with an arbitrary coordinate system. The secret is to understand how PostScript translates your drawing commands into actual path drawing on a page.

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 Windows computer it is apparently always at the lower left, but on a Unix machine frequently at the upper left. The basic units of length are the width and the height of one **pixel** (one horizontal, the other vertical). This makes sense, because in the end every drawing merely colours certain pixels on your screen or printer page in one of various colours.

The second is the **page** coordinate system. This is the one you start up with, in which as before 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 drawing in. 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. This is the same as if we had done **`306 396 translate`** before we scaled to inches.

At all times, PostScript maintains explicitly a formula for changing from user to physical coordinates, and implicitly one to change from user to page coordinates as well. The formula involves 6 numbers, and looks like

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

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

Coordinate changes like this are called **affine** coordinate transformations. One good way to write an affine coordinate transformation formula is in terms of a matrix:

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

The 2×2 matrix is called the **linear** component of the coordinate transformation, and the vector added on is called its **translation** component.

Affine transformations are characterized by the property that they take lines to lines. They also have the stronger property that they take parallel lines to parallel lines.

It might be useful to track how things go as a program proceeds. As has been mentioned, when PostScript starts up we have

$$\begin{aligned} x_{\text{page}} &= x_{\text{user}} \\ y_{\text{page}} &= y_{\text{user}} \end{aligned}$$

If we perform **`306 396 translate`** we then have

$$\begin{aligned} x_{\text{page}} &= x_{\text{user}} + 306 \\ y_{\text{page}} &= y_{\text{user}} + 396 \end{aligned}$$

If we now perform **`72 72 scale`** we have

$$\begin{aligned} x_{\mathrm{page}} &= 72\,x_{\mathrm{user}} + 306 \\ y_{\mathrm{page}} &= 72\,y_{\mathrm{user}} + 396 \end{aligned}$$

If we now put in **`90 rotate`** we have

$$\begin{aligned} x_{\mathrm{page}} &= -72\,y_{\mathrm{user}} + 306 \\ y_{\mathrm{page}} &= 72\,x_{\mathrm{user}} + 396 \end{aligned}$$

4. How PostScript stores coordinate transformations

The data determining an affine coordinate change

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

are stored in PostScript in an **array** $[a\ c\ b\ d\ e\ f]$ of size six, which it calls a **matrix**. 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

The **current transformation matrix** or **CTM** holds data giving the current transformation from user to physical coordinates. For example, we might try this at the beginning of a program and get

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

This is a typical example of what we might get on a Windows computer when starting up. It means that the origin is at lower left, with each point 4/3 of a pixel in width and height. The way this works is that the command **`matrix`** just puts an empty array on the stack and **`currentmatrix`** puts the current value of a etc. into it.

`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, the sequence below has the same effect as that above

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

Here, M is a transformation 'matrix' — an array of 6 numbers. As always, PostScript does things backwards. It applies transformations from left to right, whereas in mathematics it is more conventional to apply right to left. Of course both are just arbitrary conventions.

`M setmatrix`	Sets the current transformation matrix equal to M

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

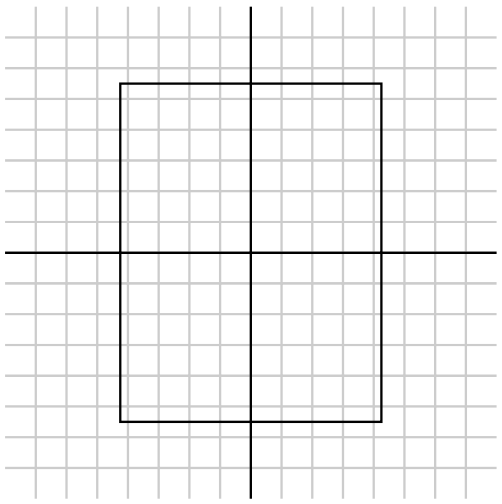
```
/user-to-page-matrix {
matrix currentmatrix
matrix defaultmatrix
matrix invertmatrix
matrix concatmatrix
} def
```

5. Picturing the coordinate system

In trying to understand how things work, it might be helpful to show some pictures of the two coordinate systems, the user's and the page's, in different circumstances.

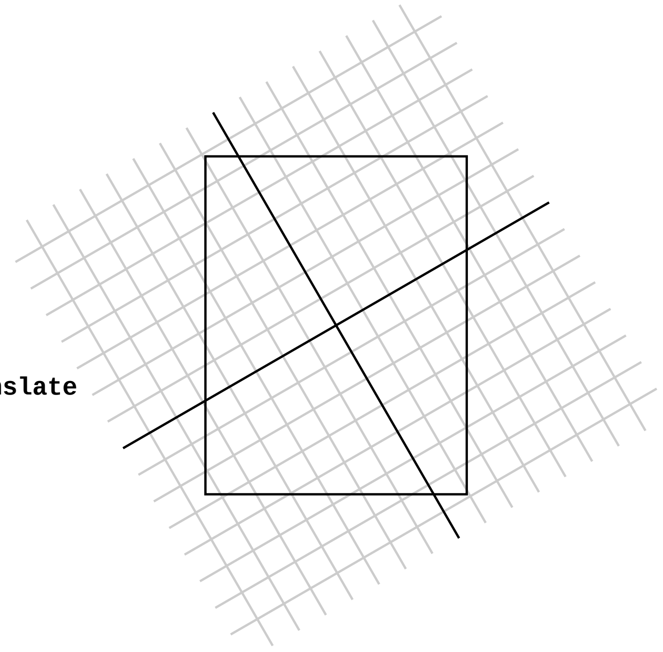
The basic geometric property of an affine transformation is that it takes parallelograms to parallelograms, and so does its inverse. Here are several pictures of how the process works. On the left in each figure is a square grid in user coordinates, on the right the image on the page, as well as the page outline itself. Also on the left is the **inverse image** of the page outline.

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

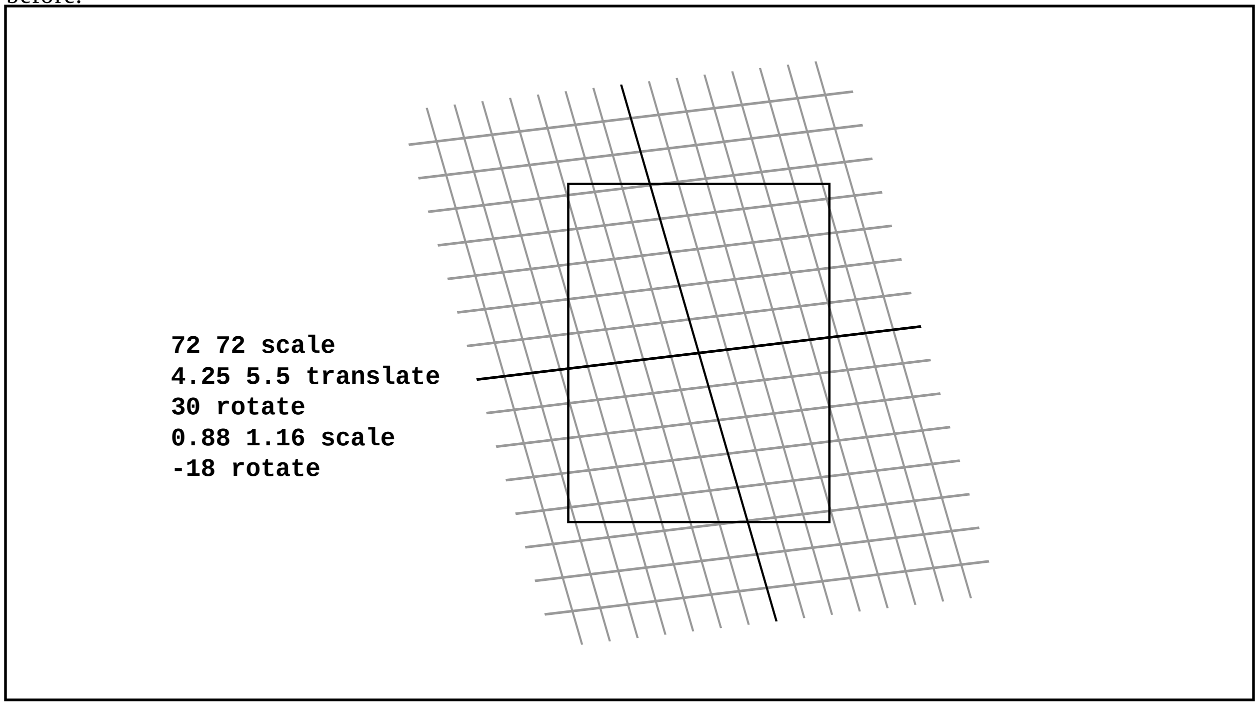


In this figure, the user unit is one inch, and a grid at that spacing is drawn at the right. Note that there are no actual lengths in user space. In all these pictures, the choice of scaling in user space is arbitrary.

```
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.



Even 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.

6. 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. The main point is that the affine transformation

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

can be rewritten as

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

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

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

are essentially affine transformations in two dimensions. The main advantage of this association is that if we perform two affine transformations successively

$$\begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} x_\bullet \\ y_\bullet \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$
$$\begin{bmatrix} x_\bullet \\ y_\bullet \end{bmatrix} \mapsto \begin{bmatrix} x_{\bullet\bullet} \\ y_{\bullet\bullet} \end{bmatrix} = \begin{bmatrix} a_\bullet & c_\bullet \\ b_\bullet & d_\bullet \end{bmatrix} \begin{bmatrix} x_\bullet \\ y_\bullet \end{bmatrix} + \begin{bmatrix} e_\bullet \\ f_\bullet \end{bmatrix}$$

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

$$\begin{bmatrix} a_\bullet & c_\bullet & e_\bullet \\ b_\bullet & d_\bullet & f_\bullet \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} .$$

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

Furthermore, the equation of the line

$$Ax + By + C = 0$$

can be expressed purely in terms of matrix multiplication as

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

This makes it simple to answer the following question:

- *Suppose we perform an affine transformation*

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

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.

The equation we start with is

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

We have

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

therefore

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

if

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

To summarize:

- *If we change coordinates according to the formula*

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

  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

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

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

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

as you can check by multiplying. Also that

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

Here is a PostScript procedure which has four arguments, a 'matrix' M and three numbers A, B, and C, which returns on the stack the three numbers $A_\bullet$, $B_\bullet$, $C_\bullet$ which go in the equation for the transform under M of the line $Ax + By + C = 0$.

```
/transform-line {
/transform-line {
8 dict begin
/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
```

In order to understand this, you should know that (1) the items in a PostScript array are numbered starting with 0, and that (2) if A is an array in PostScript, then **`A i get`** returns the i-th element of A.

Exercise 6.1. *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 6.2. *If we set*

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

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

Exercise 6.3. *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 $x + y = 1$?

7. Drawing lines at last

Here is the final routine we want:

```
/make-line {
8 dict begin
/C exch def
/B exch def
/A exch def

% save the CTM we are using now
/ctm matrix currentmatrix def
user-to-page-matrix
A B C
transform-line
% revert temporarily to the default CTM
matrix defaultmatrix setmatrix
make-line-default
% restore the old CTM
ctm setmatrix

end
} def
```

You need to know here that **`make-line-default`** is the routine alluded to in the first section that draws lines in the default coordinate system.

Exercise 7.1. *Finish the unfinished procedures you need, and assemble all the pieces into one collection of procedures that will include this main procedure. Exhibit some examples of how things work.*

8. How coordinate changes are made

To each of the basic coordinate changing commands corresponds a 3×3 matrix:

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

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

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

The effect of applying one of these commands is to multiply the current transformation matrix on the right 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 right by

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

You will rarely want to do this. Normally a combination of rotations and scales will do what you want.