Chapter 4. Coordinates and conditionals in PostScript

We will 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.

1. Coordinates

When you write a command that leads toward drawing a line, such as `0 0 moveto`, PostScript has to translate this into something that eventually makes marks somewhere, either on your computer monitor or on a piece of paper. Since you may have inserted a number of commands before this that have changed the coordinate system, such as `scale` or `translate` or `rotate`, PostScript has to keep track of the relationship between the coordinates you enter and the ones it draws with. It does this by means of data that tell it how to transform coordinates from those which you are using to draw into those where marks are made.

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 Windows computer 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 the width and the height of one **pixel** (one horizontal, the other vertical). A pixel is essentially the smallest mark that the physical device can deal with. 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 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 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 internally a formula for changing from user to physical coordinates, and implicitly one to change from user to page coordinates as well. The formula involves six numbers, and looks like

$$x_{\text{physical}} = ax_{\text{user}} + cy_{\text{user}} + e$$
$$y_{\text{physical}} = bx_{\text{user}} + dy_{\text{user}} + f$$

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. One good way to write an affine coordinate transformation formula 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.

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. Linear transformations have in addition the property that they take the origin to itself. The following can be rigourously proven, but the proof will not be given here:

- *An affine transformation of the plane takes lines to lines and 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 **perspective transformations**. These are not built into PostScript as affine transformations are.

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

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

If we now put in `90 rotate` we have

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

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 shall look at arrays in more detail in the next chapter.) 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 of 6 zeroes on the stack, and `currentmatrix` stores the current transformation matrix entries in this array. The way this works might seem a bit strange, but its purpose is probably to restrict 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 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.

`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
```

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 be the default matrix we get at start-up. The transformation from current user coordinates to the original one is therefore the matrix product CD^{-1}.

Exercise 2. How can you find out in PostScript how big a pixel (the physical device unit) is?

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 cordinates 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`.

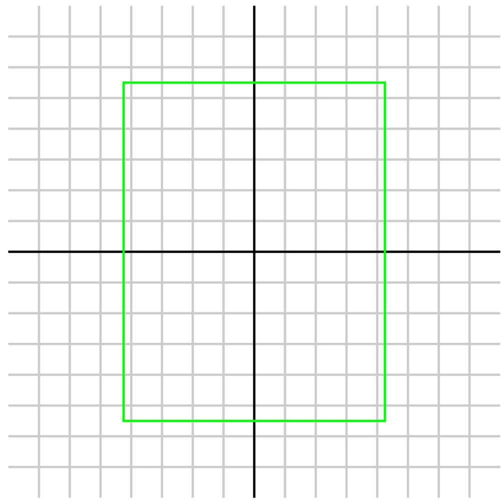
Exercise 3. Write a procedure `page-to-user` with two arguments `x y` which returns on the stack the user coordinates of the point whose page coordinates are `x y`.

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.)

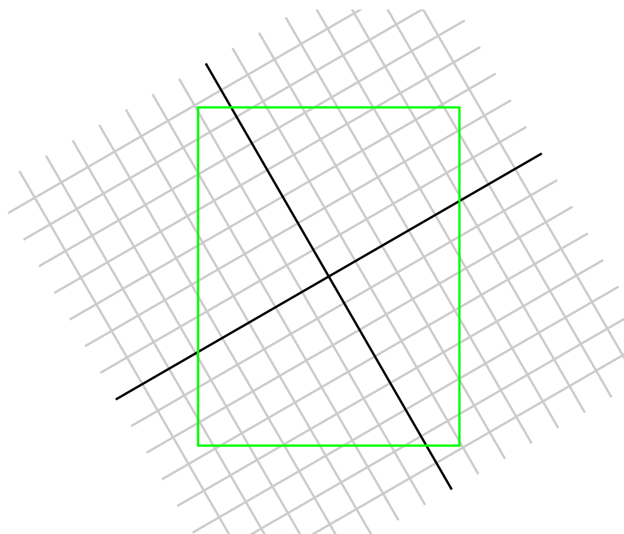
The basic geometric property of an affine transformation is that it takes parallelograms to parallelograms, and so does its inverse. Here are several pictures of how the process works. On the left in each figure is a sequence of commands, and on the right is how the resulting coordinate grid lies over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

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



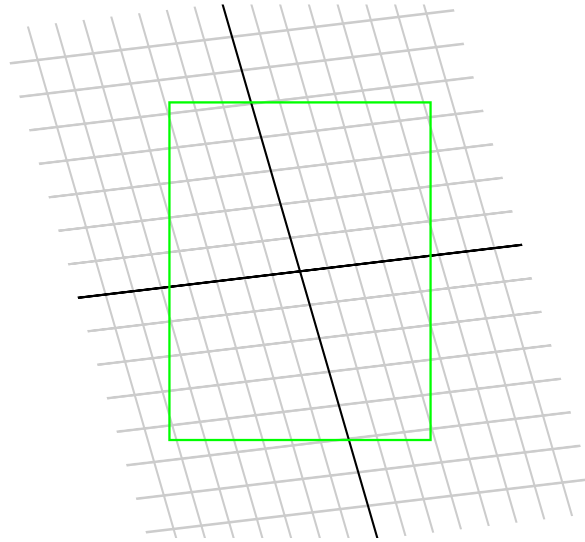
In this figure, the user unit is one inch, and a grid at that spacing is drawn at the right.

```
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
```



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.

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 essentially shifting it up one unit in height. The main point is that the *affine* 2D transformation

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

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. In other words, 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

$$[\,x \quad y\,] \mapsto [\,x_1 \quad y_1\,] = [\,x \quad y\,] \begin{bmatrix} a & b \\ c & d \end{bmatrix} + [\,e \quad f\,]$$

$$[\,x_1 \quad y_1\,] \mapsto [\,x_2 \quad y_2\,] = [\,x_1 \quad y_1\,] \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix} + [\,e_1 \quad f_1\,]$$

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, too. A big one involves calculating the effect of coordinate changes on the equations of lines. The equation of the line

$$Ax + By + C = 0$$

can be expressed purely in terms of matrix multiplication as

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

This makes it simple to answer the following question:

- *Suppose we perform an affine coordinate change*

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

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

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

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 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 {
  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
```

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

Exercise 4. 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 5. 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. 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 $x + y = 1$?

5. How coordinate changes are made

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

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

The effect of applying one of these commands is to multiply the current transformation matrix on the **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 and scales will do what you want.

Exercise 7. After operations

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

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

6. Drawing infinite lines: conditionals in PostScript

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

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

This 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 convert into

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

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 an $8.5'' \times 11''$ page. We want to design a procedure with three arguments A, B, C, whose effect is to draw the part of the line $Ax + By + C = 0$ 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 `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 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 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, too. But we will postpone this for now.

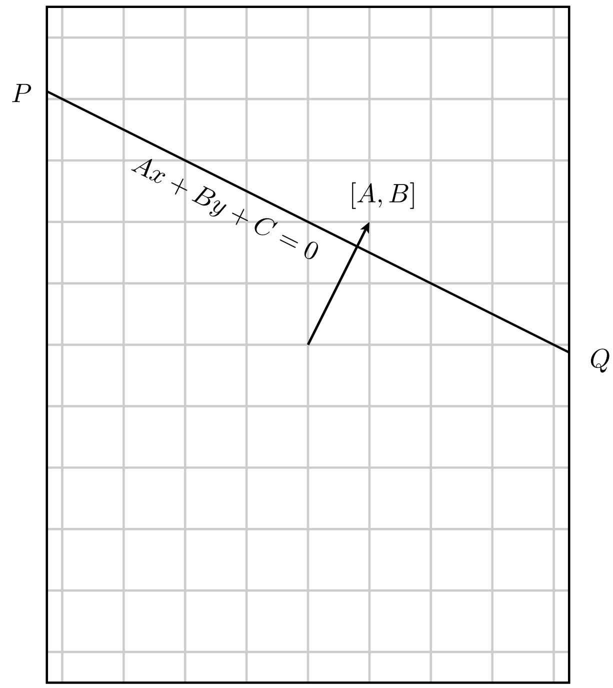
Here is the rough idea of our approach. We will divide the problem into two cases: (1) that where the line is 'essentially' horizontal; (2) that where it is 'essentially' vertical. We could in fact divide the cases according to what is suggested by our initial discussion—i.e. according to whether or 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 'essentially horizontal' and 'essentially vertical'. 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 'effectively' 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 'almost horizontal' if $|A| \le |B|$, otherwise 'almost vertical'.*

Recall that our coordinate system is in inches, centred on the page. The left hand side of the page is therefore at $x_{\mathrm{left}} = -4.25''$, the right one at $x_{\mathrm{right}} = 4.25''$. 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 $|A| \le |B|$ for a horizontal line, we must have $B \ne 0$ as well. Therefore we can solve for y, given x:

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

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.

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

```
/mkline {
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 8. Fill in the `...` to get a working procedure. Demonstrate it with a few samples.

Exercise 9. Modify the procedure above to one called `mkline-default` that works with the **default** coordinate system, the one with the origin at bottom left, unit of length one point.

7. Drawing lines in general circumstances

The routine `mkline` defined above works only in a particular coordinate system. We would now like to make up one that works no matter what the user coordinates are. So we are looking for a procedure with three arguments A, B, and C that builds in the current coordinate system, no matter what it may be, a line segment including all of the visible page. Here is the final routine we want:

```
/mkline {
  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
  mkline-default
  % restore the old CTM
  ctm setmatrix

  end
} def
```

The routine `mkline-default` called here is the routine alluded to in the first section that draws lines in the default coordinate system.

Exercise 10. 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. 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
-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 with suitable 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. 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.