

## Chapter 7. Drawing curves automatically

The process of drawing curves by programming each one specially is too complicated to be done easily. In this chapter we shall see how to construct procedures to help out enormously. We proceed in stages, starting with a reasonably simple example.

### 1. Drawing an hyperbola

The curves we have drawn so far are really too simple to give you an idea of how useful Bezier curves are. This is because the Bezier curve is guaranteed to match exactly the graph of any polynomial of degree 3 or less, so no approximation is involved. In this section we will develop a procedure `hyperbola` with these properties: • It has three arguments— $x_0$ ,  $x_1$ ,  $N$ . • It constructs the graph of the upper branch of an hyperbola

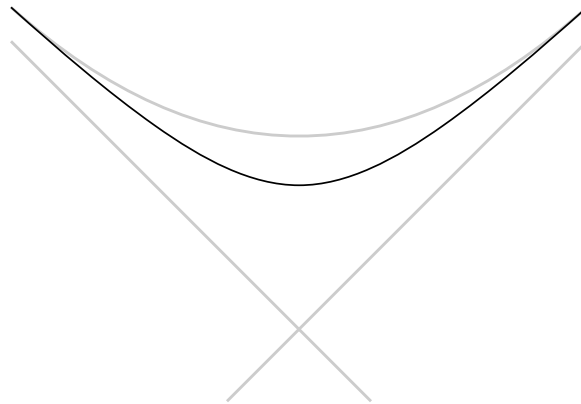
$$y^2 - x^2 = 1, \quad y = \sqrt{1 + x^2}.$$

by using  $N$  Bezier segments in between.

As is usually the best idea, the procedure builds the path without drawing it. Thus the sequence

```
newpath
-2 2 4 hyperbola
stroke
```

will draw the curve  $y = \sqrt{1 + x^2}$  from  $x = -2$  to  $x = 2$  in 4 Bezier segments.



I have added the asymptotes to convince you it really is an hyperbola, and I have also shown in light grey what you get with `-2 2 1 hyperbola` so you can see that several Bezier segments are necessary to do a good job.

What goes into the procedure `hyperbola`? We can immediately write down the skeleton

```
/hyperbola {
16 dict begin
/N exch def
/x1 exch def
/x0 exch def
```

...

```
end
} def
```

and we must now fill in the real computation. First we set a step size  $h = (x_1 - x_0)/N$ :

```
/h x1 x0 sub N div def
```

Then we introduce variables  $x$  and  $y$  which are going to change as the procedure runs, and move to the first point on the graph. It will also help to keep a variable  $s$  to hold the current value of the slope. Note that if

$$y = f(x) = \sqrt{1 + x^2} = (1 + x^2)^{1/2}$$

then

$$s = f'(x) = (1/2)(2x)(1 + x^2)^{-1/2} = \frac{x}{\sqrt{1 + x^2}} = \frac{x}{y}.$$

```
/x x0 def
/y 1 x x mul add sqrt def
/s x y div def
x y moveto
```

Now we must build  $N$  Bezier segments, using a **repeat** loop.

```
N {
  x 0.33333 h mul add
  y h 0.33333 mul s mul add
  /x x h add def
  /y 1 x x mul add sqrt def
  /s x y div def
  x h 0.33333 mul sub
  y h 0.33333 mul s mul sub
  x y
  curveto
} repeat
```

and that's it.

We could make this program somewhat more readable and more flexible if we add to it a couple of procedures which calculate  $f(x)$  and  $f'(x)$ , instead of doing it in line. Each of the procedures should have a single argument  $x$ . They are short enough that we do not need to use a variable inside them. Explicitly:

```
% sqrt(1 + x^2)

/f {
  dup mul 1 add sqrt
} def

% x/sqrt(1 + x^2)

/df {
  dup dup mul 1 add sqrt div
} def
```

Recall that `dup` just duplicates the item on the top of the stack. The new loop would be

```

N {
  x 0.33333 h mul add
  y h 0.33333 mul s mul add
  /x x h add def
  /y x f def
  /s x df def
  x h 0.33333 mul sub
  y h 0.33333 mul s mul sub
  x y
  curveto
} repeat

```

It would be better to have a single procedure that calculates  $f(x)$  and  $f'(x)$  all in one package. For one thing, it would be more efficient since we wouldn't have to calculate square roots more often than necessary. For another, I prefer to have things that go together . . . well, go together. The calculation of  $f(x)$  and  $f'(x)$  are related, and if you modify one to draw some different graph, then you will have to modify the other. For this reason they should be dealt with in one visible and indivisible unit. We can do this by using a single procedure with one argument  $x$  and as output an array of two numbers  $[y\ s]$ . But now a variable to hold the value of  $x$  is useful. It might be a good idea here to exhibit all procedures we are using:

```

% x -> [sqrt(1 + x^2), x/sqrt(1 + x^2)]

/f {
  2 dict begin
  /x exch def
  /y 1 x x mul add sqrt def
[
  y
  x y div
]
end
} def

% x0 x1 N

/hyperbola {
16 dict begin
/N exch def
/x1 exch def
/x0 exch def

% h = (x1 - x0)/N
/h x1 x0 sub N div def

/x x0 def
/F x f def
/y F 0 get def
/s F 1 get def

x y moveto

N {
  x 0.33333 h mul add
  y h 0.33333 mul s mul add

```

```

/x x h add def
/F x f def
/y F 0 get def
/s F 1 get def
x h 0.33333 mul sub
y h 0.33333 mul s mul sub
x y
  curveto
} repeat

end
} def

```

It is true that using a dictionary and an array has made the program somewhat less efficient than it was at the start. On the good side, the program is now perhaps a bit more readable—or perhaps not, depending probably on your own taste. It has one great virtue, however—it is a great deal more flexible. *If we want to draw some other graph, we need only to rewrite the single procedure  $\mathbf{f}$ , making sure that it, too, has a single argument  $x$  and returns an array of two values  $[y\ s]$ .*

**Exercise 1.1.** *Modify the procedure  $\mathbf{f}$  so you can use essentially the same stuff to graph the function  $y = x^3 - x$  between  $-1$  and  $1$ .*

**Exercise 1.2.** *Modify the procedure  $\mathbf{f}$  so you can use essentially the same stuff to graph the function  $y = \sin x$  between  $-1$  and  $1$ . Be careful about degrees and radians. Use it to draw the graph of  $\sin x$  between  $x = 0$  and  $x = \pi$  with 1, 2, 4 segments, all on one plot. Make them successively lighter so you can distinguish them.*

## 2. Parametrized curves

Not all curves in the plane are the graphs of functions. But almost any curve we can imagine is the union of a number of smooth segments, where each segment is a **parametrized path** in the plane.

Recall that a parametrized path is a pair of functions  $(x(t), y(t))$  defined for the parameter  $t$  in some given range, and that as  $t$  varies over the range we get a one dimensional curve of points in the plane. It is useful to keep in mind here a mildly important distinction between the **curve** and the **path**. The curve is a geometrical object—the thing you see—while the (parametrized) path is a particular way of describing points on the curve. If the curve is the unit circle around the origin, for example, we can think of it as the union of two graphs

$$y = \pm\sqrt{1 - x^2}$$

as  $x$  varies from  $-1$  to  $1$ , and we can also think of it as the curve traversed by the points

$$(\cos t, \sin t)$$

as  $t$  ranges from  $0$  to  $2\pi$ .

The best way to think of a parametrized path is to think of the number  $t$  as representing time, and to picture a point moving along the path as  $t$  increases. The position at time  $t$  is  $(x(t), y(t))$ . At time  $t + h$  we are at position  $(x(t + h), y(t + h))$ , and the velocity at time  $t$  is therefore

$$\lim_{h \rightarrow 0} ((x(t + h) - x(t))/h, (y(t + h) - y(t))/h) = (x'(t), y'(t)) .$$

To draw a parametrized curve we apply to each coordinate the procedure above. Suppose we are given endpoints  $(x_0, y_0)$  and  $(x_1, y_1)$ , as well as velocity vectors  $(x'_0, y'_0)$  and  $(x'_1, y'_1)$ . Then we get cubic polynomials  $P_x(t)$  and

$P_y(t)$  determined by the endpoint data. The path  $(P_x(t), P_y(t))$  will be a cubic approximation to the true curve. It can be drawn in PostScript in a way similar to that described above for graphs.

We associate to each of the coordinates  $x$  and  $y$  some interpolated points

$$\begin{bmatrix} x_{1/3} \\ y_{1/3} \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + \frac{\Delta t}{3} \begin{bmatrix} x'_0 \\ y'_0 \end{bmatrix}, \quad \begin{bmatrix} x_{2/3} \\ y_{2/3} \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} - \frac{\Delta t}{3} \begin{bmatrix} x'_1 \\ y'_1 \end{bmatrix}$$

These are the control points of the cubic segment.

And now the way you draw this cubic segment in PostScript is by the sequence

```
newpath
x[0] y[0] moveto
x[1/3] y[1/3]
x[2/3] y[2/3]
x[1] y[1]
curveto
```

where I have written `x[0]` for  $x_0$  etc.

Let's look again at the problem posed at the beginning of this section, and see how we would draw a parametrized path by using Bezier curves. We must first divide it up into a certain number of segments. Calculate the position and velocity at the endpoints of each of the segments. Move to the first point and add one Bezier curve for each segment. (The command `moveto` is not necessary in the intermediate curves, because drawing a Bezier curve in PostScript advances the current point to the end-point.) Then stroke it or fill it or clip it (if in the latter cases it is closed).

One curious fact is that although PostScript has a special command for drawing arcs of circles, behind the scenes it draws an arc as a sequence of Bezier curves! Internally, every path in PostScript is stored as a sequence of commands `moveto`, `lineto`, `curveto`, and `closepath`, along with accompanying data specified in device coordinates. A path is built without delay as you build it with commands like these in your program—the conversion from user to device coordinates takes place at the earliest moment possible. For example, if you type `0 0 moveto` the point `0 0` is converted when the command `moveto` is encountered, since it is just there that PostScript knows that `0 0` are the coordinates of a point. A path disappears when you start up a new one with `newpath`. (There are some magical tricks PostScript can perform by working with the internal representation of a path. If you want to know more about it, you can begin by looking up the PostScript command `pathforall`.)

### 3. Drawing graphs automatically

In the next section I'll explain a technique for drawing parametrized curves which will become a standard trick in your bag of tools. There are several new ingredients in it, and it may help if I explain one of them by improving the procedure for drawing hyperbolas along the same lines. We have already made that procedure reasonably flexible by isolating how the actual function  $f(x)$  is used to draw the graph. What I will do now is show how *you can change functions by using the graphing function as an argument to a general procedure for making graphs*. The *only* part of the procedure `hyperbola` which must be changed is the very first part. Since the procedure no longer draws only hyperbolas, its name must be changed. And it has an extra argument, the **name** of  $f$ . This must be a procedure into which you put  $x$  and out of which you get an array  $[f(x) f'(x)]$ . So we must read this fourth argument, which in fact I make the first of four, and we must convert that name into a procedure so we can call it. The few lines of this procedure where it differs from the older one are here:

```

/mkgraph {
16 dict begin
/N exch def
/x1 exch def
/x0 exch def
/f exch cvx def

```

Very simple. Here is a complete program which uses this to draw the graph of  $y = x^4$ .

```

% /f x0 x1 N

/mkgraph {
16 dict begin
/N exch def
/x1 exch def
/x0 exch def
/f exch cvx def

% h = (x1 - x0)/N
/h x1 x0 sub N div def

/x x0 def
/F x f def
/y F 0 get def
/s F 1 get def

x y moveto

N {
  x 0.33333 h mul add
  y h 0.33333 mul s mul add
  /x x h add def
  /F x f def
  /y F 0 get def
  /s F 1 get def
  x h 0.33333 mul sub
  y h 0.33333 mul s mul sub
  x y
  curveto
} repeat

end
} def

% [x^4 4x^3]

/quartic {
  2 dict begin
  /x exch def
[
x x mul x mul x mul
x x mul x mul 4 mul
]
end

```

```

} def

% -----

72 72 scale
4.25 5.5 translate
0.012 setlinewidth

newpath
/quartic -1 1 8 mkgraph
stroke

```

In the next section I shall introduce a procedure which is rather similar to this one. It differs in these aspects: (1) It deals with parametrized curves instead of graphs. (2) It allows you to use a single procedure to draw any one of a large family of curves, for example all of the graphs  $y = cx^4$  where  $c$  is a constant you can specify when you draw the curve. (3) It adds the new path to the path that already exists, if there is one.

**Exercise 3.1.** Write a PostScript procedure with the same arguments as `mkgraph` but which simply draws a polygon among the successive points. (This can be used to debug your calculus.)

**Exercise 3.2.** Write a PostScript procedure that will graph a polynomial between  $x_0$  and  $x_1$  with  $N$  Bezier segments. There are a number of things you have to think about: (1) For evaluating a polynomial in a program it is easiest to use an expression like  $5x^3 + 2x + 3x + 4 = ((5x + 2)x + 3)x + 4$ . (2) You will have to add an argument to this procedure to pass the polynomial coefficients as an array. Recall that `length` returns the size of an array.

#### 4. Drawing parametrized paths automatically

If you are given a parametrized path and you want to draw it by using Bezier curves, you must calculate position and velocity at several points of the path. This is tedious and prone to error, and you will quickly ask if there is some way to get PostScript to do the work. This is certainly possible, if you can write a PostScript routine which calculates position and velocity for the parametrization. One tricky point is that we don't want to rewrite the drawing routine for every path we draw, but would like instead to put in the **parametrization** itself as an argument passed to the routine. The parametrization should be a procedure which has a single argument, namely a value of  $t$ , and returns data giving position and velocity for that value of  $t$ . We shall in fact do this, and make the name of the routine which calculates position and velocity one of the arguments. How PostScript handles this is somewhat technical, but you won't have to understand underlying details to understand the routine. Another tricky point is that we would like to make a routine for drawing a family of curves—we don't want to have to make separate routines for  $y = x^2$ ,  $y = 2x^2$ , etc. We would like to be able at least to write one routine that can draw  $y = cx^2$  for any specified constant  $c$ . This will be accomplished by passing to the routine an array of **parameters** to pick a specific curve from a family. The terminology is a bit clumsy: we have the parameters determining which path from a family is to be drawn and the variable  $t$  which parametrizes the curve. The last, to avoid confusion, I shall call the **parametrizing variable**.

I will first simply lay out the main routine we shall use from now on for drawing parametrized paths. It is a bit complicated. In the next two sections I shall explain how to use it and how it works.

---

```

% stack: [parameters] /f t0 t1 N

/mkpath {
16 dict begin
/N exch def
/t1 exch def
/t0 exch def
/f exch cvx def
/pars exch def

/h t1 t0 sub N div def
/h3 h 0.333333 mul def

/currentloc pars t0
f def
pars t0 f 0 get
aload pop

thereisacurrentpoint
{
  lineto
}
{
  moveto
}
ifelse

N {
  currentloc 0 get 0 get      % x y = currentpoint
  currentloc 1 get 0 get      % x0 dx0
  h3 mul
  add

  currentloc 0 get 1 get
  currentloc 1 get 1 get
  h3 mul
  add

  /t0 t0 h add def
  /currentloc pars t0 f def
  currentloc 0 get 0 get
  currentloc 1 get 0 get
  h3 mul sub
  currentloc 0 get 1 get
  currentloc 1 get 1 get
  h3 mul sub
  currentloc 0 get 0 get
  currentloc 0 get 1 get
  curveto
} repeat
end
} def

```



We shall assume that this code is in a file called `mkpath.inc` (for *inclusion*).

## 5. How to use it

The input to the procedure consists of five items on the stack.

- First comes an array `[ ... ]` of parameters, which I shall say something more about in a moment. It can be just the empty array `[]`, but it must be there.
- Then follows the name of a routine specifying the parametrization in PostScript. A name in PostScript starts with the symbol `/`. As I have already said, this routine has two arguments. The first is an array of things which the routine can use to do its calculation. The second is the variable `t`. Its output (left on the stack) will be a  $2 \times 2$  matrix written according to my conventions as an array of two 2D arrays.
- The third is the initial value of `t` for the path.
- The fourth is the final value of `t`.
- Fifth is the number `N` of Bezier segments to be drawn.

The most important, and complicated, item here is the parametrization routine.

For example, suppose we want to draw circles of varying radii, centred at the origin. The parametrization of such a circle is

$$t \mapsto P(t) = (R \cos t, R \sin t)$$

and the velocity vector of this parametrization is

$$t \mapsto P'(t) = (-R \sin t, R \cos t)$$

if `t` is in radians. The variable `t` is the parameter which traverse the path, while `R` is a parameter specifying which of several possible circles are to be drawn. Thus the input to the circle drawing routine will be a pair `[R] t` and output will be `[[xt yt][dxt dyt]]`.

Here is a more explicit block of PostScript code (assuming  $\pi$  defined elsewhere). Note that PostScript uses degrees instead of radians, but that the formula for the velocity vector assumes radians, and we must convert from radians to degrees.

```
/circle {
4 dict begin
/t exch def
/pars exch def
/R pars 0 get def
/t t 180 mul pi div def
[
[ t cos R mul t sin R mul ]
[ t sin neg R mul t cos R mul ]
]
end
} def
```

```
(mkpath.inc) run
```

```
newpath
[2] /circle 0 2 pi mul 8 mkpath
closepath
stroke
```

This example shows among other things that the routine `mkpath` does not actually draw a path, but builds it just as the PostScript `curveto` does, and allows the programmer to decide what to do with it—to stroke it or fill it, for example, or even clip to it. In fact, the routine `mkpath` does something a bit more complicated. *It adds to any path already drawn, by beginning with a straight line from the last point drawn to to the start of the path segment about to be built.* This is the way the PostScript `arc` behaves, and it seems a good model to follow. At any rate, it is for this reason that the routine tests whether or not there already exists a current point. The overall effect is to allow you to build up a series of paths so as to make a rather complicated one. You might even draw several successive segments of the same path, for example, using a different value of the integer  $N$  (the fifth argument) to allow for parts of the path being more or less straight.

Another possibly interesting point is how to pass a procedure to another procedure. What we do in fact is to pass the *name* of the procedure (begun with `/`) to our routine, and the routine converts the name to a procedure with the command `cvx`.

The array of parameters can be of any size you want. It can even be empty if in fact you just want to draw one of a kind. But you will probably find that most paths you want to draw are just part of a larger family.

There is one other point. It is very common to get the formula for the velocity vector wrong, and usually you will draw rather wild curves instead of the correct one. There is a simple way to see if this is the case: you can use a routine that might be called `mkpolypath` which simply draws a polygonal path instead of one made up of Bezier segments. It has exactly the same usage as `mkpath`, but ignores the velocity vector in building the path, and can hence be used to see if the rest of your routine is working.

**Exercise 5.1.** Write a procedure `mkpolypath` which has the same arguments as `mkpath` but draws a polygon instead.

**Exercise 5.2.** Write down a parametrization of the ellipse

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$$

Write a procedure which will draw it. For example you would write `3 4 drawellipse`.

**Exercise 5.3.** Draw the image of the  $12 \times 12$  grid centred at  $(0, 0)$ , under the map  $(x, y) \mapsto (x^2 - y^2, 2xy)$ . The spacing between grid lines is 0.25 cm.

## 6. How it works

The basic idea is simply to automate the procedure you used earlier to draw the graph of  $y = x^4$ . The point is that if we are given a parametrization of a path we can draw an approximation by Bezier curves using the velocity vectors associated to the parametrization in order to construct control points. The routine is fairly straightforward, except that it calls a procedure `thereisacurrentpoint` to tell whether the path being drawn is the beginning of a new path or the continuation of an old one. You don't have to know the details of the procedure called — it operates in a very simple manner at a somewhat low level of PostScript.

This routine all by itself is very useful, and is capable of making interesting pictures. In combination with a programmed version of the chain rule from second year calculus it is capable of producing spectacular ones. We shall see later how the basic routine `mkpath` can be incorporated in others, for example to draw three dimensional parametrized paths in perspective.

thales  
eudoxus  
euclid  
rchimede