CHAPTER 7

Drawing curves automatically: procedures as arguments

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.

7.1. Drawing an hyperbola

The curves we have drawn so far are really too simple to give you an idea of how useful Bézier curves are. This is because the Bézier 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 three arguments—x_0, x_1, N—that constructs the graph of the upper branch of an hyperbola

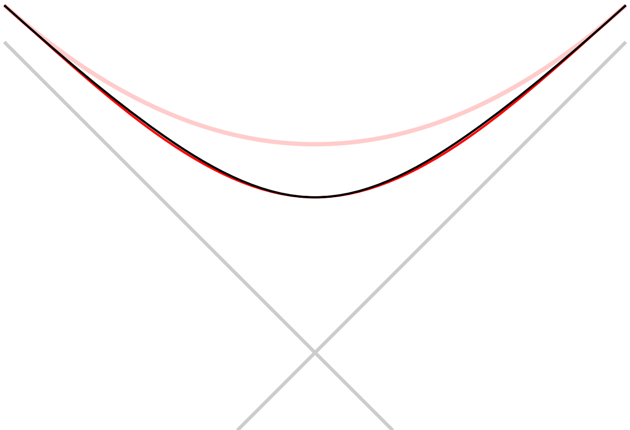
$$y^2 - x^2 = 1, \quad y = \sqrt{1 + x^2}\ .$$

by using N Bézier segments in between x_0 and x_1.

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 Bézier segments.



Paths drawn by `-2 2 1 hyperbola` *(pink) and* `-2 2 2 hyperbola` *(red).*

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 $h = (x_1 - x_0)/N$ so that in N steps we cross from x_0 to x_1:

```
/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

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

Recall that the control points are

$$\begin{aligned}
&(x_0, y_0) \\
&(x_0 + h/3, y_0 + y'(x_0)/3 \\
&(x_1 - h/3, y_1 - y'(x_1)/3 \\
&(x_1, y_1) \ .
\end{aligned}$$

where $x_1 = x + h$, $y_1 = y(x_1)$. The code begins:

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

Now we must build N Bézier segments, using a `repeat` loop.

```
N { % repeat
  x h 3 div add
  y h 3 div s mul add
  /x x h add def
  /y 1 x x mul add sqrt def
  /s x y div def
  x h 3 div sub
  y h 3 div 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 that 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)
/f' {
  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 { % repeat
  x h 3 div add
  y h 3 div s mul add
  /x x h add def
  /y x f def
  /s x f' def
  x h 3 div sub
  y h 3 div 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 {  % repeat
    x h 3 div add
    y h 3 div s mul add
    /x x h add def
    /F x f def
    /y F 0 get def
    /s F 1 get def
```

```
    x h 3 div sub
    y h 3 div 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* `f`*, making sure that it, too, has a single argument* x *and returns an array of two values* $[y\ s]$.

Exercise 7.1. *Modify the procedure* `f` *so you can use essentially the same stuff to graph the function* $y = x^3 - x$ *between* -1 *and* 1.

Exercise 7.2. *Modify the procedure* `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—it is only when* x *is expressed in radians that* $\sin' x = \cos x$. *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.*

7.2. Parametrized curves

We now have a good idea of how to draw smooth function graphs. However, not all curves in the plane are the graphs of functions. What is true is that almost any curve we can imagine is the union of a number of smooth segments, where each segment is a parametrized path $(x(t), y(t))$ in the plane.

I recall that to draw by Bézier curves a segment between $t = t_0$ and t_1 I use control points

$$
\begin{aligned}
P_0 &= (x_0, y_0) \\
&= (x(t_0), y(t_0)) \\
P_1 &= (x_1, y_1) \\
&= (x(t_1), y(t_1)) \\
P_{1/3} &= \begin{bmatrix} x_{1/3} \\ y_{1/3} \end{bmatrix} \\
&= \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + \frac{\Delta t}{3} \begin{bmatrix} x_0' \\ y_0' \end{bmatrix} \\
P_{2/3} &= \begin{bmatrix} x_{2/3} \\ y_{2/3} \end{bmatrix} \\
&= \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} - \frac{\Delta t}{3} \begin{bmatrix} x_1' \\ y_1' \end{bmatrix} .
\end{aligned}
$$

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 Bézier 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 Bézier curve for each segment. (The command `moveto` is not necessary in the intermediate curves, because drawing a Bézier 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).

7.3. 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 itself 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 the function f, which must be a procedure into which you put x and out of which you get an array $[f(x)\ f'(x)]$. We must read this fourth argument and 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 {
  load
  16 dict begin
  /f exch def
  /N exch def
  /x1 exch def
  /x0 exch def
```

Not quite as simple as might be expected. A slight technical difficulty is caused by the fact that the name of the parametrization procedure being used here might be redefined inside the procedure—it might, for example, be `/f`—so it must be retrieved from its dictionary with `load` before a new dictionary is introduced. Here is a complete program which uses this to draw the graph of $y = x^4$.

```
 % x0 x1 N /f
/mkgraph { load
  /f exch def
  1 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 h 3 div add
    y h 3 div s mul add
    /x x h add def
    /F x f def
    /y F 0 get def
    /s F 1 get def
    x h 3 div sub
     y h 3 div 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
1 72 div setlinewidth

newpath
-1 1 8 /quartic mkgraph
stroke
```

In the next section we'll see a procedure which is rather similar to this one, but 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. Like the command `arc`.

Exercise 7.3. *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 7.4. *Write a PostScript procedure that will graph a polynomial between x_0 and x_1 with N Bézier 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. (See Appendix 6 for more about polynomial evaluation.)*

7.4. Drawing parametrized paths automatically

If you are given a parametrized path and you want to draw it by using Bézier 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:  t0 t1 N [parameters] /f
/mkpath { load
1 dict begin
/f exch def
/pars exch def
/N exch def
/t1 exch def
/t0 exch def
   % h = (t1-t0)/N
  /h t1 t0 sub N div def
   % h3 = h/3
  /h3 h 3 div def
   % set current location = [f(t0) f'(t0)]
  /currentloc pars t0 f def
  pars t0 f 0 get
  aload pop  % calculate the first point
  thereisacurrentpoint  % if a path already under construction ...
    { lineto }
    { moveto }
  ifelse
  N {                            % x y = currentpoint
    currentloc 0 get 0 get       % x0 dx0
    currentloc 1 get 0 get
    h3 mul
    add
     % x1
    currentloc 0 get 1 get
    currentloc 1 get 1 get
    h3 mul
    add
     % y1
    /t0 t0 h add def
     % move ahead one step
    /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
     % x2 y2
    currentloc 0 get 0 get
    currentloc 0 get 1 get
     % x3 y3
    curveto
  } repeat
end } def
```

The procedure `thereisacurrentpoint` returns true or false, depending on whether a path has been started. We have already seen it in Chapter 6.

7.5. How to use it

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

- *The first is the initial value of t for the path.*
- *The second is the final value of t.*
- *The third is the number N of Bézier segments to be drawn.*
- *Fourth 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.*
- *Last 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 2×2 matrix written according to my conventions as an array of two $2D$ arrays.*

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

For example, suppose we want to draw circles of varying radii, centered 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 to be π). Note that PostScript uses degrees instead of radians, but that the formula for the velocity vector assumes radians.

```
/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
```

You might not have seen anything like `//pi` before—using `//` before a variable name means the interpreter puts its current value in place immediately, rather than wait until the procedure is run to look up `pi` and evaluate it.

The array of parameters passed to one of these routines 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 routine for the velocity vector wrong, since after all you must first calculate a derivative. When this happens, the curve will hit the points representing position, but will wander wildly in between. One way to see if in fact you have computed the derivative correctly is to use a routine that might be called `mkpolypath` which simply draws a polygonal path instead of one made up of Bézier 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 7.5. *Write a procedure* `mkpolypath` *which has the same arguments as* `mkpath` *but draws a polygon instead.*

Exercise 7.6. *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 7.7. *Draw the image of the* 12×12 *grid centered at* $(0, 0)$*, under the map* $(x, y) \mapsto (x^2 - y^2, 2xy)$*. The spacing between grid lines is to be* 0.25 *cm.*

7.6. 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 Bézier 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. But the ideas behind it are applicable in a wide range of circumstances.

7.7. Code

The file `mkpath.inc` contains a procedure `mkpath` as well as `mkgraph` and a few other related ones.