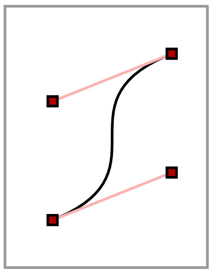
CHAPTER 6

Curves



So far, the only paths we have learned how to draw in PostScript are sequences of line segments. It is possible to assemble a good approximation of just about any curve by a large number of segments, but there are more elegant and efficient ways, involving **Bézier curves**.

6.1. Arcs

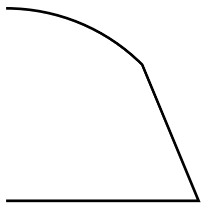
The simplest curves are circles. There are two special commands to draw circles and pieces of circles. The sequence

```
0 0 10 47 67 arc
```

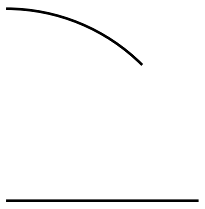
will add to the current path the short arc of a circle of radius 10, centerd at the origin, between arguments 47° and 67°. If `arcn` is used it will draw the clockwise arc around the long way, instead (`arcn` for **arc n**egative).

It would be a good idea to investigate here what 'adding to the current path' means, because it is behavior many of our later procedures will imitate. Here are two short sketches that should illustrate how it works. In the first, we draw a line and then continue drawing an arc. The default behavior is for an arc to continue the current path in this way—to add a line from the last point of the previous path to the first point of the arc. Sometimes this is not what one wants or expects, in which case it is necessary to add a `moveto` to break up the path, as in the second figure.

```
newpath
0 0 moveto
1 0 lineto
0 0 1 45 90 arc
stroke
```



```
newpath
0 0 moveto
1 0 lineto
45 cos 45 sin moveto
0 0 1 45 90 arc
stroke
```



Another curiousity of `arc` is that in a coordinate system in which y-units are distinct from x-units it produces an ellipse. In other words, it always draws the locus of an equation $(x-a)^2 + (y-b)^2 = r^2$ in user coordinates (x, y). If the axes are not perpendicular or the x and y units are different, it will look like an ellipse.

6.2. Fancier curves

Lines and arcs of circles make up a very limited repertoire. PostScript allows a third method to build paths, which is much more versatile. In creating complicated paths, for example the outlines of characters in a font, this third method is indispensable.

Conceptually, the simplest way to draw even a complicated curve is by drawing a sequence of line segments—that is to say, making a **polygonal approximation** to it—but this usually requires a very large number of segments to be at all acceptable. It also suffers from the handicap that it is not very scalable—that is to say, even if a collection of segments looks smooth at one scale, it may not look good at another. Here, for example, is a portion of the graph of $y = x^4$ drawn with eight linear segments.

The overall shape is not too bad, but the breaks are quite visible. You can certainly improve the quality of the curve by using more segments, but then the number of segments required to satisfy the eye changes with the scale used in representing the curve. One trouble is that the human eye can easily perceive that the directions vary discontinuously in this figure. This is not at all something to be taken for granted—the more we learn about vision in nature the more we learn that most features like this depend on sophisticated image processing. In contrast, the human eye apparently has trouble perceiving discontinuities in curvature.

In any event, it is better, if possible, to produce smooth curves—at least as smooth as the physical device at hand will allow. PostScript does this by approximating segments of a curve by **Bézier cubic curves**. This allows us to have the tangent direction of a curve vary continuously as well.

The Bézier curve is the last major ingredient of PostScript to be encountered. In the rest of this book we shall learn how to manipulate and combine the basic tools we have already been introduced to.

6.3. Bézier curves

In PostScript, to add a curved path to a path already begun, you put in a command sequence like

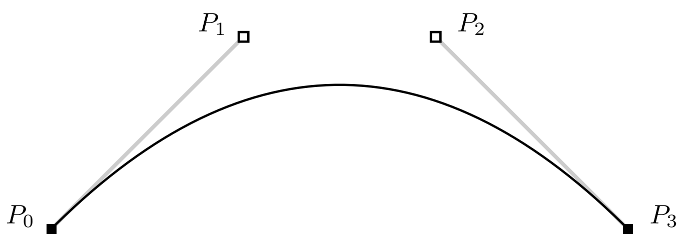
```
1 1
2 1
3 0
curveto
```

This makes a curve starting at the **current point** P_0, ending at $P_3 = (3, 0)$, and in between following a path **controlled** by the intermediate points $P_1 = (1, 1)$ and $P_2 = (2, 1)$. If there is no current point, a `moveto` command should precede this. Thus:

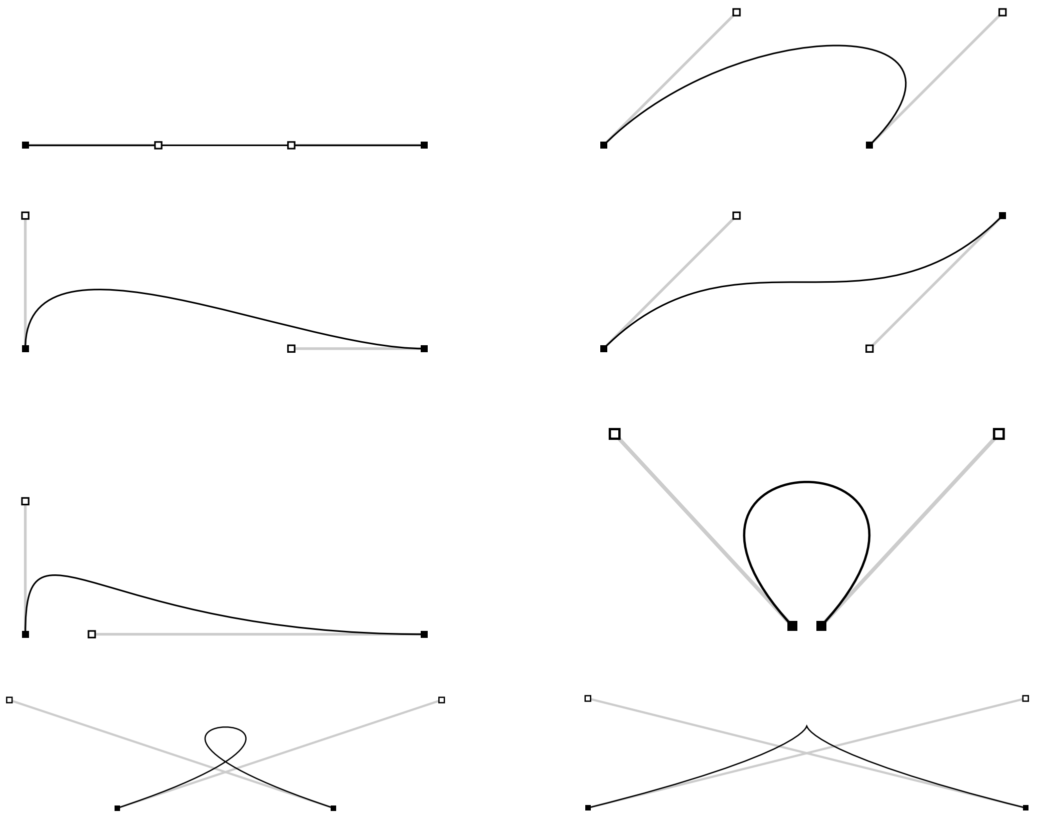
```
0 0 moveto
1 1
2 1
3 0
curveto
```

would make a complete curve starting at $(0, 0)$.

In short, `curveto` behaves very much like `lineto` but depends on a larger set of points. At any rate, what we get is this (where the four relevant points are marked):



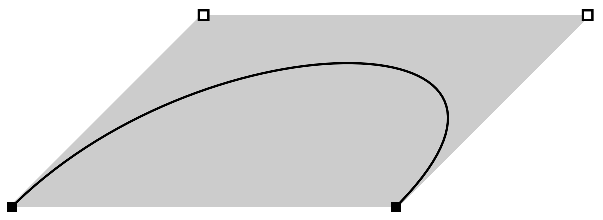
In this book I shall usually call P_0 and P_3 the **end points** and P_1 and P_2 the **control points** of the curve. Occasionally I shall just call all of them control points, which is more standard terminology. Even from this single picture you will see that the effect of the control points on the shape of the curve is not so simple. In order to draw curves efficiently and well, we have to understand this matter much better. We shall see later the exact mathematics of what is going on, but right now I shall simply exhibit several examples.



You should be able to see from these examples that the use of control points to specify curves becomes intuitive with experience. The following facts may give a rough feeling for how things go.

(1) *The path starts at P_0 and ends at P_3.*
(2) *When the curve starts out from P_0 it is heading straight for P_1.*
(3) *Similarly, when it arrives at P_3 it is coming from the direction of P_2.*
(4) *The longer the line from P_0 to P_1, the tighter the curve sticks to that line when it starts out from P_0. Similarly for P_2 and P_3.*

There is another fact that is somewhat less apparent.



(5) *If we wrap up the four points P_i in a quadrilateral box, then the whole curve is contained inside that box.*

The intuitive picture of a Bézier curve conceives of it as a path followed by a particle in motion between certain times. The vector from P_0 to P_1 is proportional to the velocity of the particle as it starts out from P_0, and the vector from P_2 to P_3 is proportional to its velocity when it arrives at P_3. Another way of putting this is that roughly speaking the control points are a convenient way to encode the initial and final velocities in geometric data. This explains properties (2), (3), and (4). Property (5) is implied by the fact that any point on the curve is some kind of weighted average of the four points P_i, as we shall see later.

Curves drawn by using control points in this way are called **Bézier curves** after the twentieth century French automobile designer Pierre Bézier who was one of the very first to use them extensively in computer graphics, even though their use in mathematics under the name of **cubic interpolation curves** is much older.

One natural feature of Bézier curves described by control points is that they are stable under arbitrary affine transformations—that is to say that the affine transformation of a Bézier curve is the Bézier curve defined by the affine transformations of its control points. This is often an extremely useful property to keep in mind.

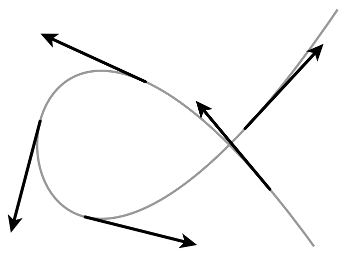
Exercise 6.1. *Write a PostScript procedure* `pixelcurve` *with arguments* 4 *arrays* P_0, P_1, P_2, P_3 *of size* 2, *with the effect of drawing the corresponding Bézier curve, including also black pixels of width* $0.05''$ *at each of these points.*

6.4. How to use Bézier curves

In this section we shall be introduced to a recipe for using Bézier curves to draw very general curves. In the next this recipe will be justified. In order to make the recipe plausible, we shall begin by looking at the problem of how to approximate a given curve by polygons.

The first question we must answer, however, is more fundamental: *How are curves to be described in the first place?* In this book the answer will usually be in terms of a **parametrization**. Recall that a **parametrized curve** is a map from points of the real line to points in the plane—that is to say, to values of t in a selected range we associate points $(x(t), y(t))$ in the plane. It often helps one's intuition to think of the **parameter** t as time, so as time proceeds we move along the curve from one point to another. In this scheme, with a parametrization $P(t)$, the **velocity** vector at time t is the limit of average velocities over smaller and smaller intervals of time $(t, t+h)$:

$$V(t) = P'(t) = \lim_{h \to 0} \frac{P(t+h) - P(t)}{h} = [x'(t), y'(t)] \ .$$



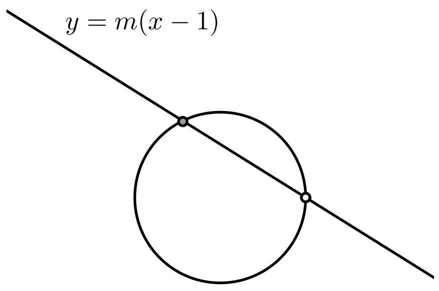
The direction of the velocity vector is tangent to the curve, and its magnitude is determined by the speed of motion along the curve.

Example. The unit circle with center at the origin has parametrization $t \mapsto (\cos t, \sin t)$.

Example. If $f(x)$ is a function of one variable x, its graph has the parametrization $t \mapsto (t, f(t))$.

In other words, a parametrization is essentially just a pair of functions $(x(t), y(t))$ of a single variable, which is called the parameter. The parameter often has geometric significance. For example, in the parametrization $t \mapsto (\cos t, \sin t)$ of the unit circle it is the angle at the origin between the positive x-axis and the radius to the point on the circle.

Example. Besides the standard parametrization of the circle there is another interesting one. If ℓ is any line through the point $(1, 0)$ other than the vertical line $x = 1$, it will intersect the circle at exactly one other point on the circle.



The equation of such a line will be $y = m(x-1) = mx - m$, where m is its slope. The condition that a point $(x, mx-m)$ lie on the circle is

$$\begin{aligned} x^2 + y^2 &= 1 \\ &= x^2 + m^2(x-1)^2 \\ &= x^2(1+m^2) - 2m^2x + m^2 \\ x^2 - 2\frac{m^2}{m^2+1}\,x + \frac{m^2-1}{m^2+1} &= 0 \\ (x-1)\left(x - \frac{m^2-1}{m^2+1}\right) &= 0 \end{aligned}$$

so that

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

As m varies from $-\infty$ to ∞ the point (x, y) traverses the whole circle except the point $(1, 0)$. Thus m is a parameter, and

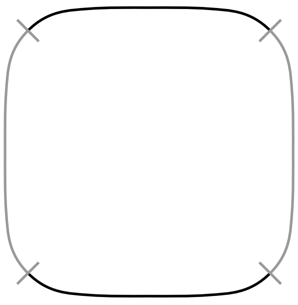
$$m \longmapsto \left(\frac{m^2 - 1}{m^2 + 1}, \frac{-2m}{m^2 + 1} \right)$$

a kind of parametrization of the unit circle. This has historical significance. If we set $m = p/q$ to be a fraction, the point (x, y) will be a point on the unit circle with rational coordinates, say $(a/c, b/c)$ with $(a/c)^2 + (b/c)^2 = 1$. If we clear denominators, we obtain a set of three integers a, b, c with $a^2 + b^2 = c^2$. Such a set is called a **Pythagorean triple**. There is evidence that this construction was known to the Babylonians in about 1800 B.C.

Exercise 6.2. *Use this idea to find the smallest several Pythagorean triples.*

Example. There are two common ways to specify a curve in the plane. The first is a parametrization. The second is an equation relating x and y. An example is the oval

$$x^4 + y^4 = 1 \; .$$



This is not the graph of a function, and it has no obvious single parametrization. We can solve the equation $x^4 + y^4 = 1$ to get

$$y = \sqrt[4]{1 - x^4}$$

which gives the top half of our oval, and get the bottom half similarly. Neither half is yet the graph of a good function, however, because both have infinite slope at $x = \pm 1$. We can, however, restrict the range of x away from ± 1, say to $[-\sqrt[4]{1/2}, \sqrt[4]{1/2}]$. We can then turn the curve sideways and now solve for x in terms of y to write the rest as a graph rotated 90°. To summarize, we can at least express this curve as the union of four separate pieces, each of which we can deal with.

Exercise 6.3. *Find a parametrization of this oval by drawing inside it a circle, and taking as the point corresponding to t the point of intersection of the oval with the ray from the origin at angle t.*

Exercise 6.4. *Sketch the curve $y^2 = x^2(x + 1)$ by hand in the region $|x| \le 3$, $|y| \le 3$. Find a parametrization of this curve by using the fact that the line $y = mx$ will intersect it at exactly one point other than the origin. Write down this parametrization. Use it to redo your sketch in PostScript, in any way that looks convincing, to check your drawing.*

With this understanding of how a curve is given to us, the question we are now confronted with is this:

- *Given a parametrization $t \mapsto P(t)$ of a curve in the plane, how do we draw part of it using Bézier curves?*

If we were to try to draw it using linear segments, the answer would go like this: Suppose we want to draw the part between given values t_0 and t_1 of t. We divide the interval $[t_0, t_1]$ into n smaller intervals $[t_0+ih, t_0+(i+1)h]$, and then draw lines $P(t_0)P(t_0+h)$, $P(t_0+h)P(t_0+2h)$, $P(t_0+2h)P(t_0+3h)$, etc. Here $h = (t_1 - t_0)/n$. If we choose n large enough, we expect the series of linear segments to approximate the curve reasonably well.

To use Bézier curves, we will follow the roughly the same plan—chop the curve up into smaller pieces, and on each small piece attempt to approximate the curve by a single Bézier curve. In order to do that, the essential problem we face is this: *Suppose we are given two values of the parameter t, which we may as well assume to be t_0 and t_1, and which we assume not to be too far apart. How do we approximate by a single Bézier curve the part of the curve parametrized by the range $[t_0, t_1]$?*

Calculating the end points is no problem. But how to get the two interior control points? Since they have something to do with the directions of the curve at the end points, we expect to use the values of the velocity vector at the endpoints. The exact recipe is this. Start by setting

$$\begin{aligned} P_0 &= (x(t_0), y(t_0)) \\ P_3 &= (x(t_1), y(t_1)) \ . \end{aligned}$$

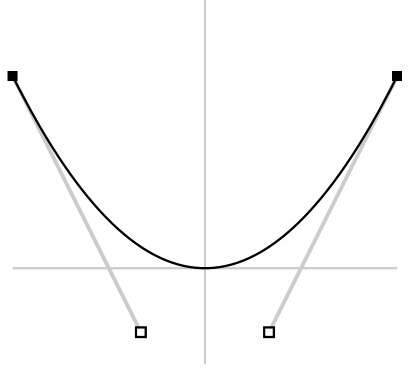
These are the end points of our small Bézier curve. Then set

$$\begin{aligned} \Delta t &= t_1 - t_0 \\ P_1 &= P_0 + (\Delta t/3)P'(t_0) \\ P_2 &= P_3 - (\Delta t/3)P'(t_1) \end{aligned}$$

to get the control points.

Example. Let's draw the graph of the parabola $y = x^2$ for x in $[-1, 1]$. It turns out that a single Bézier curve will make a perfect fit over the whole range. Here the parametrization is $P(t) = (t, t^2)$, $P'(t) = [1, 2t]$.

$$\begin{aligned} t_0 &= -1 \\ t_1 &= 1 \\ \Delta t &= 2 \\ P_0 &= (-1, 1) \\ P_1 &= (1, 1) \\ P'(-1) &= (1, -2) \\ P'(1) &= (1, 2) \\ P_1 &= P_0 + (2/3)P'(t_0) \\ &= (-1/3, -1/3) \\ P_2 &= (1/3, -1/3) \end{aligned}$$

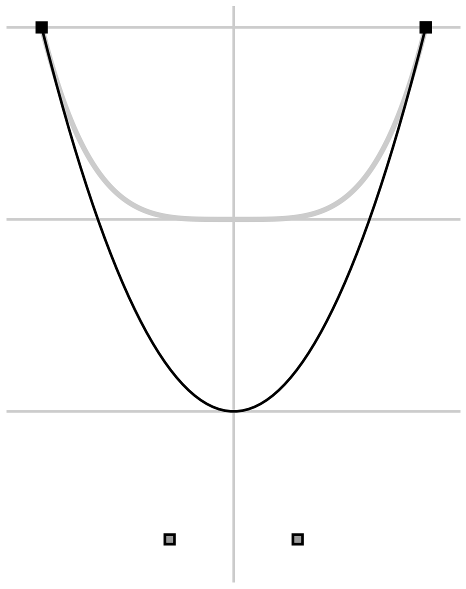


Example. Let's draw the graph of $y = x^4$ for $x = -1$ to $x = 1$. Here $P(t) = (t, t^4)$, $P'(t) = (1, 4t^3)$. We shall do this with 1, 2, and 4 segments in turn.

(a) One segment $[-1], 1]$. We have this table, with the control points interpolated.

x	y	x'	y'
-1.0000	1.0000	1.0	-4.0
-0.3333	-1.6667		
0.3333	-1.6667		
1.0000	1.0000	1.0	4.0

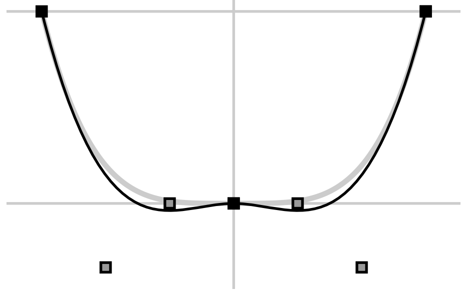
The approximation is foul. Droopy.



(b) Two segments $[-1, 0]$ and $[0, 1]$.

x	y	x'	y'
-1.0000	1.0000	1.0	-4.0
-0.6667	-0.3333		
-0.3333	0.0000		
0.0000	0.0000	1.0	0.0
0.3333	0.0000		
0.6667	-0.3333		
1.0000	1.0000	1.0	4.0

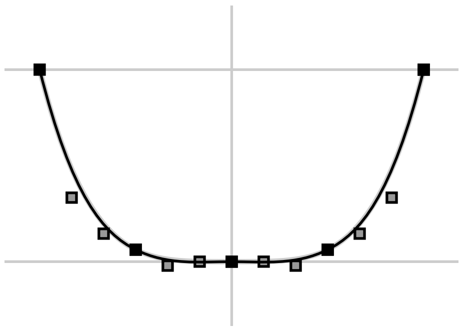
Somewhat better.



(c) Four segments $[-1.0, -0.5]$, $[-0.5, 0.0]$, $[0.0, 0.5]$, $[0.5, 1.0]$.

x	y	x'	y'
-1.0000	1.0000	1.0	-4.0
-0.8333	0.3333		
-0.6667	0.1458		
-0.5000	0.0625	1.0	-0.5
-0.3333	-0.0208		
-0.1667	0.0000		
0.0000	0.0000	1.0	0.0
0.1667	0.0000		
0.3333	-0.0208		
0.5000	0.0625	1.0	0.5
0.6667	0.1458		
0.8333	0.3333		
1.0000	1.0000	1.0	4.0

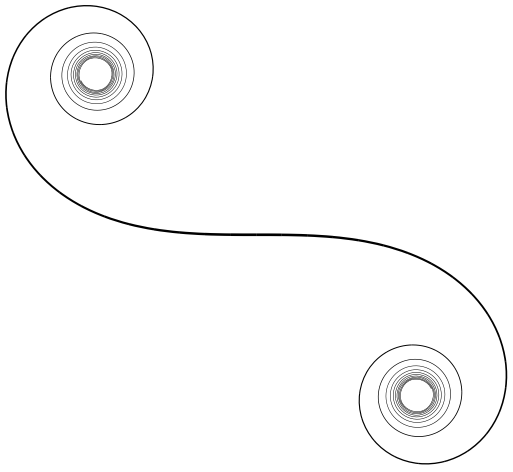
It is almost indistinguishable from the true graph. It is perhaps only when you see where the control points lie that you notice the slight rise in the middle.



Exercise 6.5. *In many situations, drawing a parametrized path by Bézier curves, using the velocity vector to produce control points, is more trouble than it's worth. This is true even if the procedure is to be automated somewhat as explained in the next chapter, since calculating the velocity can be quite messy. There is one situation in Bézier plotting is definitely the method of choice, however, and that is when the path is given by a path integral. The Cornu spiral, for example, is the path in the complex plane defined by*

$$C(t) = \int_0^t e^{-is^2}\, ds$$

as t ranges from $-\infty$ to ∞. In this case, $C(t)$ can only be approximated incrementally by numerical methods, say by Simpson's rule, but the velocity $C'(t)$ comes out of the calculation at no extra cost, since it is just the integrand. Furthermore, evaluating $C(t)$ will be expensive in effort since each step of the approximation involves some work, so the fewer steps taken the better.



Plot the Cornu spiral, which is shown above, using Bézier curves. One subtle point in this figure is that the thickness of the path above decreases as the curve spirals further in, because otherwise the spirals would be clotted.

Exercise 6.6. *The remarks in the previous exercise are just as valid for the plots of first order differential equations in the plane by numerical methods. Plot using Bézier curves the trajectories of*

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

starting at a few uniformly distributed points around the unit circle.

6.5. The mathematics of Bézier curves

The mathematical problem we are looking at in drawing good curves in computer graphics is that of approximating an arbitrary parametrized path $t \mapsto (x(t), y(t))$ by a simpler one. If we are approximating a path by line segments, for example, then we are replacing various pieces of the curve between points $P_0 = P(t_0)$ and $P_1 = P(t_1)$ by a linearly parametrized path

$$t \mapsto \frac{(t_1 - t)P_0 + (t - t_0)P_1}{(t_1 - t_0)}$$

from one point to the other. This parametrization can be better understood if we write this as

$$t \mapsto (1 - s)P_0 + sP_1 \text{ where } s = \frac{t - t_0}{t_1 - t_0} .$$

With Bézier curves, we are asking for a parametrization from one point to the other with the property that its coordinates are cubic polynomials of t (instead of linear). In other words, we are looking for approximations to the coordinates of a parametrization by polynomials of degree three. We expect an approximation of degree three to be much better than a linear one.

The Bézier curve, then, is to be a parametrized path $B(t)$ from P_0 to P_3, cubic in the parameter t, and depending in some way on the interior control points P_1 and P_2. Here it is:

$$\begin{aligned} B(t) &= \frac{(t_1 - t)^3 P_0 + 3(t - t_1)^2 (t - t_0) P_1 + 3(t_1 - t)(t - t_0)^2 P_2 + (t - t_0)^3 P_3}{(t_1 - t_0)^3} \\ &= (1 - s)^3 P_0 + 3(1 - s)^2 s P_1 + 3(1 - s) s^2 P_2 + s^3 P_3 \quad \big(s = (t - t_0)/(t_1 - t_0)\big) . \end{aligned}$$

We shall justify this formula later on. The form using s is easier to calculate with than the other, as well as more digestible.

It is simple to verify that

$$\begin{aligned} B(t_0) &= P_0 \\ B(t_1) &= P_3 \,. \end{aligned}$$

We can also calculate (term by term)

$$\begin{aligned} (t_1 - t_0)^3 B'(t) &= -3(t_1 - t)^2 P_0 + 3(t_1 - t)^2 P_1 - 6(t - t_0)(t_1 - t) P_1 \\ &\qquad + 6(t - t_0)(t_1 - t) P_2 - 3(t - t_0)^2 P_2 + 3(t - t_0)^2 P_3 \\ B'(t) &= \frac{3(t_1 - t)^2 (P_1 - P_0) + 6(T - t_0)(t - t_1)(P_2 - P_1) + 3(t - t_0)^2 (P_3 - P_2)}{(t_1 - t_0)^3} \\ B'(t_0) &= \frac{3(P_1 - P_0)}{t_1 - t_0} \\ B'(t_1) &= \frac{3(P_3 - P_2)}{t_1 - t_0} \end{aligned}$$

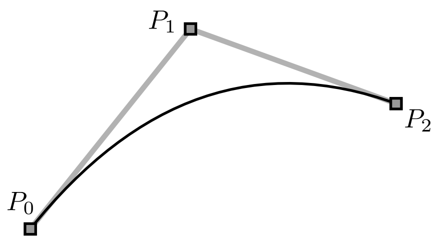
These calculations verify our earlier assertions relating the control points to velocity, since we can deduce from them that

$$\begin{aligned} P_1 &= P_0 + \left(\frac{t_1 - t_0}{3} \right) B'(t_0) \\ P_2 &= P_3 - \left(\frac{t_1 - t_0}{3} \right) B'(t_1) \,. \end{aligned}$$

6.6. Quadratic Bézier curves

A **quadratic Bézier curve** determined by three control points P_0, P_1, and P_2 is defined by the parametrization

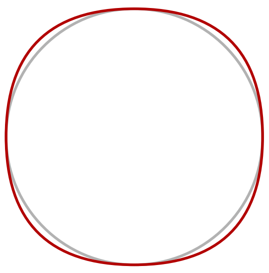
$$Q(s) = (1 - s)^2 P_0 + 2s(1 - s) P_1 + s^2 P_2 \,.$$



It is a degenerate case of a Bézier curve, with control points P_0, $(1/3)P_0 + (2/3)P_1$, $(2/3)P_1 + (1/3)P_2$, P_2, as you can easily check. But sometimes it is easy to find control points for a quadratic curve, not so easy to find good ones for a cubic curve. One good example arises in drawing implicit curves $f(x, y) = 0$. In this case, we can often calculate the gradient vector $[\partial f/\partial x, \partial f/\partial y]$ and then that of the tangent line

$$\frac{\partial f}{\partial x}(x - x_0) + \frac{\partial f}{\partial y}(y - y_0) = 0$$

at a point (x_0, y_0) on the curve. But we can approximate the curve between two points P and Q by the quadratic Bézier curve with intermediate control point the intersection of the two tangent lines at P and Q. The figure below shows how the curve $x^2 + y^2 - 1 = 0$ is approximated by four quadratic curves (in red). An approximation by eight quadratic curves is just about indistinguishable from a true circle.



6.7. Mathematical motivation

In using linear or Bézier paths to do computer graphics, we are concerned with the problem of approximating the coordinate functions of an arbitrary path by polynomials of degree one or three. Considering each coordinate separately, we are led to try to approximate an arbitrary function of one variable by a polynomial of degree one or three.

The basic difference between linear approximations and cubic approximations lies in the following facts:

- *If t_0, t_1, y_0, and y_1 are given then there exists a unique linear function $f(t)$ such that*

$$\begin{aligned} f(t_0) &= y_0 \\ f(t_1) &= y_1 \end{aligned}$$

- *Given t_0, t_1, y_0, y_1, v_0, v_1, there exists a unique cubic polynomial $f(t)$ such that*

$$\begin{aligned} f(t_0) &= y_0 \\ f'(t_0) &= v_0 \\ f(t_1) &= y_1 \\ f'(t_1) &= v_1 \ . \end{aligned}$$

Roughly speaking, with linear approximations we can only get the location of end points exactly, but with cubic approximation we can get directions exact as well.

We shall prove here the assertion about cubic functions. If

$$f(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

then the conditions on $P(t)$ set up four equations in the four unknowns a_i which turn out to have a unique solution (assuming of course that $t_0 \neq t_1$). Here are the equations:

$$\begin{array}{ccccccccc} a_0 & + & a_1 t_0 & + & a_2 t_0^2 & + & a_3 t_0^3 & = & y_0 \\ & & a_1 & + & 2a_2 t_0 & + & 3a_3 t_0^2 & = & v_0 \\ a_0 & + & a_1 t_1 & + & a_2 t_1^2 & + & a_3 t_1^3 & = & y_1 \\ & & a_1 & + & 2a_2 t_1 & + & 3a_3 t_1^2 & = & v_1 \end{array}$$

The coefficient matrix is

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ & 1 & 2t_0 & 3t_0^2 \\ 1 & t_1 & t_1^2 & t_1^3 \\ & 1 & 2t_1 & 3t_1^2 \end{bmatrix}$$

It has already been remarked that the mathematics is simplified by **normalizing** the parameter variable t, so that instead of going from t_0 to t_1 it goes from 0 to 1. This is done by defining a new parameter variable

$$s = \frac{t - t_0}{t_1 - t_0} \ .$$

Note that s takes values 0 and 1 at the ends $t = t_0$ and $t = t_1$. Changing the parameter variable in this way doesn't affect the curve traversed. It simplifies the assertion above.

- *Given y_0, y_1, v_0, v_1, there exists a unique cubic polynomial $f(t)$ such that*

$$\begin{aligned} f(0) &= y_0 \\ f'(0) &= v_0 \\ f(1) &= y_1 \\ f'(1) &= v_1 \ . \end{aligned}$$

The coefficient matrix is now

$$\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{array}$$

I leave it to you as an exercise to check now by direct row reduction that the determinant is not zero, which implies that the system of four equations in four unknowns has a unique solution. Of course we know from the formula used in the previous section what the explicit formula is, but the reasoning in this section shows that this formula is the well defined answer to a natural mathematical question.

Exercise 6.7. *What is the determinant of this 4×4 matrix?*

Exercise 6.8. *Find the coefficients a_i explicitly.*

If we put together the results of this section with those of the previous one, we have this useful characterization:

- *Given two parameter values t_0, t_1 and four points P_0, P_1, P_2, P_3, the Bézier path $B(t)$ is the unique path $(x(t), y(t))$ with these properties:*
  (1) *The coordinates are cubic as a function of t;*
  (2) *$B(t_0) = P_0$, $B(t_1) = P_3$;*
  (3) *$B'(t_0) = 3(P_1 - P_0)/\Delta t$ and $B'(t_1) = 3(P_3 - P_2)/\Delta t$, where $\Delta t = t_1 - t_0$.*

The new assertion here is uniqueness. Roughly, the idea is that four control points require eight numbers, and that the cubic coordinate functions also require eight numbers.

6.8. Weighted averages

The formula for a linear path from P_0 to P_1 is

$$\begin{aligned} P(t) &= (1 - t)P_0 + tP_1 \\ &= P_0 + t(P_1 - P_0) \ . \end{aligned}$$

We have observed before that $P_0 + t(P_1 - P_0)$ may be seen as the point t of the way from P_0 to P_1. When $t = 0$ this gives P_0, and when $t = 1$ it gives P_1. There is also an intuitive way to understand the first formula that we have not considered so far.

Let's begin with some examples. With $t = 1/2$ we get the mid-point of the segment

$$\frac{P_0 + P_1}{2}$$

which is the average of the two. With $t = 1/3$ we get the point one third of the way

$$\frac{2P_0 + P_1}{3}$$

which is to say that it is a **weighted average** of the endpoints with P_0 given twice as much weight as P_1.

There is a similar way to understand the formula for Bézier curves. It is implicit in what was said in the last section that the control points P_i determine a cubic path from P_0 to P_1

$$B(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$$

and that this path is a parametrization of the Bézier curve with these control points. In other words, $B(t)$ is a weighted combination of the control points. It is actually an average—which is to say, that the sum of the coefficients is 1:

$$(1-t)^3 + 3t(1-t)^2 + 3t^2(1-t) + t^3 = \big((1-t)+t\big)^3 = 1$$

by the binomial theorem for $n = 3$, which asserts that

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3 \ .$$

Since all the coefficients in our expression are non-negative for $0 \le t \le 1$, $B(t)$ will lie inside the quadrilateral wrapped by the control points.

This idea will now be explored in more detail.

If P_0, P_1, $\ldots P_{n-1}$ is a collection of n points in the plane, then a sum

$$c_0P_0 + c_1P_1 + \cdots + c_{n-1}P_{n-1}$$

is called a weighted average of the collection if (1) all the $c_i \ge 0$; (2) the sum of all the c_i is equal to 1. In the rest of this section, our primary goal will be to describe geometrically the set of all points we get as the weighted averages of a collection of points, as the coefficients vary over all possibilities.

If $n = 2$, we know already that the set of all weighted averages $c_0P_0 + c_1P_1$ is the same as the line segment between P_0 and P_1, since we can write $c_1 = t$, $c_0 = (1-t)$.

Suppose $n = 3$, and consider the weighted average

$$c_0P_0 + c_1P_1 + c_2P_2 \ .$$

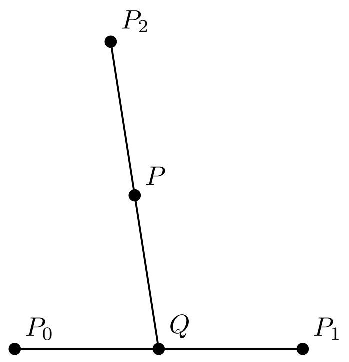
Let's look at an explicit example—look more precisely at

$$P = (1/4)P_0 + (1/4)P_1 + (1/2)P_2 \ .$$

The trick we need to carry out is to rewrite this as

$$P = (1/2)[(1/2)P_0 + (1/2)P_1] + (1/2)P_2 = (1/2)Q + (1/2)P_2$$

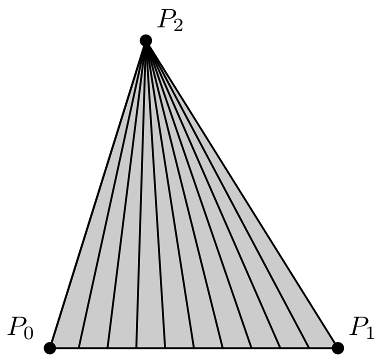
where $Q = (1/2)P_0 + (1/2)P_1$. In other words, P is the weighted average of the two points Q and P_2. The point Q is the weighted average of the original points P_0 and P_1, hence must lie on the line segment between P_0 and P_1. In other words, we have the following picture:



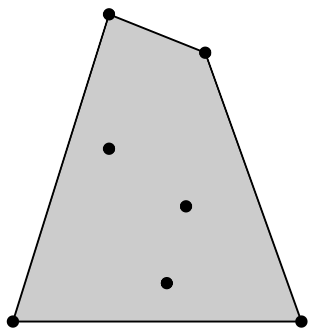
Now we can almost always perform this trick, since we can write

$$c_0P_0 + c_1P_1 + c_2P_2 = (c_0 + c_1)\left(\left(\frac{c_0}{c_0+c_1}\right)P_0 + \left(\frac{c_1}{c_0+c_1}\right)P_1\right) + c_2P_2$$

unless $c_0 + c_1 = 1 - c_2 = 0$. In the exceptional case we are just looking at P_2 itself, and in all other cases each weighted average of the three points is a weighted average of P_2 with a point on the line segment between P_0 and P_1. In other words, the set of all weighted averages of the three points coincides with the triangle spanned by the three points.



If we now look at four points, we get all the points on line segments connecting P_3 to a point in the triangle spanned by the first three. And in general we get all the points in a shape called the **convex hull** of the collection of points, which may be described very roughly as the set of points which would be contained inside a rubber band stretched around the whole collection and allowed to snap to them.



The convex hull of a set of points in the plane or in space is very commonly used in mathematical applications, and plays a major role in computational graphics as well.

Exercise 6.9. *Write the simplest procedure you can with these properties: (1) it has two arguments x_0 and x_1 and (2) it draws the graph of $y = x^2$ between x_0 and x_1 with a single Bézier curve.*

Exercise 6.10. *Draw $y = x^5$ between $x = -1$ and $x = 1$ in the same way we drew $y = x^4$ earlier.*

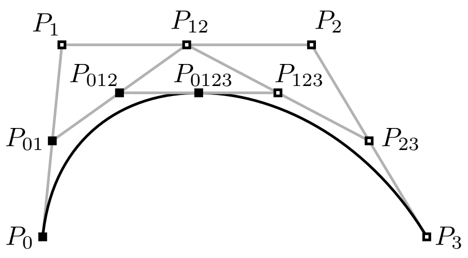
6.9. How the computer draws Bézier curves

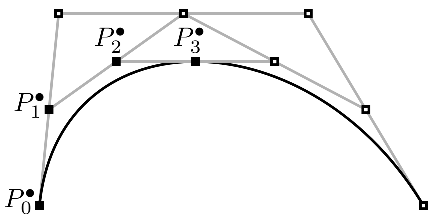
In this section we shall see how the computer goes about drawing a Bézier curve. It turns out to be an extremely efficient process. First of all, a computer 'thinks of' any path as a succession of small points (pixels) on the particular device it is dealing with. This is somewhat easier to see on a computer screen, certainly if you use a magnifying glass, but remains true even of the highest resolution printers. So in order to draw something it just has to decide which pixels to color. It does this by an elegant recursive procedure, something akin to the following way to draw a straight line segment: (1) Color the pixels at each end. (2) Color the pixel at the middle. (3) This divides the segment into two halves. Apply steps (2) and (3) again to each of the halves. And so on, until the segments you are looking at are so small that they cannot be distinguished from individual pixels.

The analogous construction for Bézier curves, attributed to the car designer de Casteljau, goes like this:

Start with a Bézier curve with control points P_0, P_1, P_2, P_3. Perform the following construction. Set

$$\begin{aligned} P_{01} &= \text{the median between } P_0 \text{ and } P_1 \\ P_{12} &= \text{the median between } P_1 \text{ and } P_2 \\ P_{23} &= \text{the median between } P_2 \text{ and } P_3 \\ P_{012} &= \text{the median between } P_{01} \text{ and } P_{12} \\ P_{123} &= \text{the median between } P_{12} \text{ and } P_{23} \\ P_{0123} &= \text{the median between } P_{012} \text{ and } P_{123} \end{aligned}$$





Then set

$$\begin{aligned} P_0^\bullet &= P_0 \\ P_1^\bullet &= P_{01} \\ P_2^\bullet &= P_{012} \\ P_3^\bullet = P_0^{\bullet\bullet} &= P_{0123} \\ P_1^{\bullet\bullet} &= P_{123} \\ P_2^{\bullet\bullet} &= P_{23} \\ P_3^{\bullet\bullet} &= P_3 \,. \end{aligned}$$

The point $P_3^\bullet$ turns out to lie on the Bézier curve determined by the original points P_i, at approximately the halfway point. The Bézier curve can now be split into two halves, each of which is itself a Bézier cubic, and the control points of the two new curves are among those constructed above. The two halves might be called the Bézier curves **derived** from the original. The points $P_i^\bullet$ are those for the first half, the $P_i^{\bullet\bullet}$ for the second. If we keep subdividing in this way we get a sequence of midpoints for the smaller segments (actually a kind of branched list), and to draw the curve we just plot these points after the curve has been subdivided far enough. This sort of subdivision can be done very rapidly by a computer, since dividing by two is a one-step operation in base 2 calculations, and in fact drawing the pixels to go on a straight line is not a great deal faster.

Exercise 6.11. *Draw the figure above with PostScript.*

Exercise 6.12. *The point $P_1^\bullet$ is $(1/2)P_0 + (1/2)P_1$. Find similar expressions for all the points constructed in terms of the original four.*

Exercise 6.13. *The purpose of this exercise is to prove that each half of a Bézier curve is also a Bézier curve. Let*

$$P(s) = (1-s)^3 P_0 + 3s(1-s)^2 P_1 + 3s^2(1-s)P_2 + s^3 P_3 \ .$$

The point is to verify that this formula agrees with the geometrical process described above. Let $P_1^\bullet$ etc. be the points defined just above. The first half of the Bézier curve we started with is a cubic curve with initial parameter 0 and final parameter $1/2$. Let $\Delta t = 1/2$. Verify that

$$\begin{aligned} P(0) &= P_0 \quad \textit{(trivial)} \\ P'(0) &= (\Delta t/3)(P_1^\bullet - P_0) \quad \textit{(almost trivial)} \\ P(1/2) &= P_3^\bullet \\ P'(1/2) &= (\Delta t/3)(P_3^\bullet - P_2^\bullet) \ . \end{aligned}$$

These equations, by the earlier characterization of control points in terms of derivatives, guarantee that the first half of the original Bézier path is a Bézier path with control points $P_0 = P_0^\bullet$, $P_1^\bullet$, $P_2^\bullet$, $P_3^\bullet$.

Exercise 6.14. *How might a computer construct quadratic Bézier curves in a similar way?*

6.10. Bernstein polynomials

The Bézier cubic polynomial

$$y_0(1-t)^3 + 3y_1(1-t)^2 t + 3y_2(1-t)t^2 + y_3 t^3$$

is just a special case of a more general construction of **Bernstein polynomials**. In degree one we have the linear interpolating function

$$y_0(1-t) + y_1 t \ ,$$

in degree two we have the quadratic functions mentioned earlier, and in degree n we have the polynomial

$$B_y(t) = y_0(1-t)^n + n y_1 t(1-t)^{n-1} + \frac{n(n-1)}{2}\, y_2\,(1-t)^{n-2}t^2 + \cdots + y_n t^n$$

where y is the array of the **control values** y_i and the other coefficients make up the n-th row of Pascal's triangle

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ & & & \cdots & & & \end{array}$$

These polynomials were first defined by the Russian mathematician Sergei Bernstein in the early twentieth century in order to answer a sophisticated question in approximation theory.

These also are weighted sums of the control values, so for $0 \le t \le 1$ the value of $B_y(t)$ will lie in the range spanned by the y_i. In particular, if the y_i are a non-decreasing sequence

$$y_0 \le y_1 \le \dots y_n$$

then $y_0 \le B_y(t) \le y_n$ for $0 \le t \le 1$. But much more can be said.

Exercise 6.15. *Prove that*

$$B'_y(t) = nB_{\Delta y}(t)$$

where Δy is the array of differences

$$\Delta y = (y_1 - y_0, y_2 - y_1, \dots, y_n - y_{n-1}) \ .$$

Exercise 6.16. *Prove that if the y_i are non-decreasing then $B_y(t)$ is a non-decreasing function over the range* $[0, 1]$.

Exercise 6.17. *There is a way to evaluate $B_y(t)$ for $0 \le t \le 1$ along the lines used by the computer to construct the Bézier cubic curve. It can be described best in a recursive fashion. First of all, if y has length 1 the Bernstein polynomial is just a constant. Otherwise, with $n > 0$, form a derived sequence of length $n - 1$:*

$$\delta y = ((1-t)y_0 + ty_1, \dots, (1-t)y_{n-1} + ty_n) \ .$$

Then

$$B_y(t) = B_{\delta y}(t) \ .$$

Prove this. Explain how this process is related to the naïve construction of Pascal's triangle, one row at a time.

6.11. This section brings you the letter O

Paths can be constructed in PostScript in various ways through commands `moveto`, etc. but internally PostScript stores a path as an array storing exactly 4 different types of objects—`moveto`, `lineto`, `curveto`, `closepath` tags together with the arguments of the command. This array can be accessed explicitly by means of the command `pathforall`. This command has four arguments, each of which is a procedure. It loops through all the components of the current path, pushing appropriate data on the stack and then applying the procedures respectively to `moveto`, `lineto`, `curveto`, and `closepath` components. For `moveto` and `lineto` components it pushes the corresponding values of x and y in current user coordinates; for `curveto` it pushes the six values of x_1, y_1, etc. (also in user coordinates); and for `closepath` it pushes nothing. The following procedure, for example, displays the current path.

```
/path-display {
  { [ 3 1 roll (moveto) ] == }
  { [ 3 1 roll (lineto) ] ==  }
  { [ 7 1 roll (curveto) ] == }
  { [ (closepath) ] == }
  pathforall
} def
```

The following procedure tells whether a current path has already been started, since it returns with `true` on the stack if and only if the current path has at least one component.

```
/thereisacurrentpoint{
  false {
    { 3 { pop } repeat true exit }
    { 3 { pop } repeat true exit }
    { 7 { pop } repeat true exit }
    { pop true exit }
  } pathforall
} def
```

The most interesting paths in PostScript are probably strings—i.e. the paths formed by strings when the `show` operator is applied, or in other words the path the string will make when it is drawn in the current font. This outline can be accessed as a path by applying the command `charpath`, which has two arguments. The first is a string. The second is a boolean variable which is more or less irrelevant to our purposes. The command appends the path described by the string in the current graphics environment to the current path, assuming in particular that a font has been selected. In this way, for example, you can deal with the outlines of strings as if they were ordinary paths. The code

```
/Times-Roman findfont
40 scalefont
setfont

newpath
0 0 moveto
(Times-Roman) false charpath
gsave
1 0 0 setrgbcolor
fill
grestore
stroke
```

produces

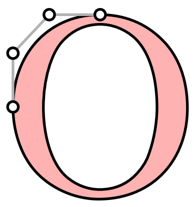
Times-Roman

You can combine `charpath` and `pathforall` to see the explicit path determined by a string, but only under suitable conditions. Many if not most PostScript fonts have a security mechanism built into them that does not allow the paths of their characters to be deconstructed, and you will get an error from `pathforall` if you attempt to do so. So if you want to poke around in character paths you must be working with a font that has not been declared inaccessible. This is not a serious restriction for most of us, since there are many fonts, including the ones usually stocked with GhostScript, that are readable. Here is the path of the character 'O' from the font called `/Times-Roman` by GhostScript:

```
0.360998541 0.673999 moveto
0.169997558 0.673999 0.039997559 0.530835 0.039997559 0.33099854 curveto
0.039997559 0.236999512 0.0697631836 0.145998538 0.119995117 0.0879980475 curveto
0.177995607 0.0249975584 0.266994625 -0.0140014645 0.354995131 -0.0140014645 curveto
0.551994622 -0.0140014645 0.689997554 0.125998542 0.689997554 0.326999515 curveto
0.689997554 0.425998539 0.660305202 0.510998547 0.603996575 0.570998549 curveto
0.540996075 0.639997542 0.456999511 0.673999 0.360998541 0.673999 curveto
closepath
0.360996097 0.63399905 moveto
0.406997085 0.63399905 0.452998042 0.618159175 0.488999 0.58999753 curveto
```

```
   0.542998075 0.540998518 0.58 0.44699952 0.58 0.328000486 curveto
   0.58 0.269001454 0.564633787 0.200002447 0.540998518 0.148002923 curveto
   0.531999528 0.123002931 0.515 0.098002933 0.491999507 0.0750024393 curveto
   0.456999511 0.0400024429 0.411999524 0.0260009766 0.358999 0.0260009766 curveto
   0.312998056 0.0260009766 0.26799804 0.04253418 0.232998043 0.0710034147 curveto
   0.180998534 0.117004395 0.15 0.218005374 0.15 0.329006344 curveto
   0.15 0.431005865 0.177197263 0.528005362 0.217998043 0.575004876 curveto
   0.256997079 0.618005395 0.30599609 0.63399905 0.360996097 0.63399905 curveto
   closepath
   0.721999526 -0.0 moveto
```

and here is the first Bézier curve in the path:



References

1. R. E. Barnhill and R. F. Riesenfeld (editors), **Computer Aided Geometric Design**, Academic Press, 1974. This book contains papers presented at a conference at the University of Utah that initiated much of modern computer graphics. The article by P. Bézier is very readable.

2. G. Farin, **Curves and surfaces for computer aided design**, Academic Press, 1988. This is a pleasant book that probably covers more about curves and surfaces than most readers of this book will want, but the first chapter is an enjoyable account by P. Bézier on the origins of his development of the curves that bear his name. These curves were actually discovered much earlier (before computers were even a dream) by the mathematicians Hermite and Bernstein, but it was only the work of Bézier, who worked at the automobile maker Renault, and de Casteljau, who worked at Citroen, that made these curves familiar to graphics specialists.

3. D. E. Knuth, METAFONT**: the Program**, Addison-Wesley, 1986. Pages 123–131 explain extremely clearly the author's implementation of Bézier curves in his program METAFONT. For the admittedly rare programmer who wishes to build his own implementation (at the level of pixels), or for anyone who wants to see what attention to detail in first class work really amounts to, this is the best resource available.