Curve approximation by splines

Victor Eijkhout

August 2004

1 Introduction to curve approximation

You may never have thought of it, but fonts (actually, typefaces) usually have a mathematical definition somehow. If a font is given as a bitmap, this is typically the result from a more compact description. Imagine the situation that you have bitmaps at 300dpi, and you buy a 600dpi printer. It wouldn't look pretty.

There is then a need for a mathematical way of describing arbitrary shapes. These shapes can also be three-dimensional; in fact, a lot of the mathematics in this chapter was developed by a car manufacturer for modeling car body shapes. But let us for now only look in two dimensions, which means that the curves are lines, rather than planes.

A mathematical formalism for curves should have these properties:

- The description should be clear and unambiguous.
- It should be easy to tinker with the shapes. This is important for the design phase.
- Constructing and evaluating the curves should be computationally cheap.
- The curves should be well behaved: small changes in the definition should not lead to unexpected bulges or spikes.
- Curves should be easily composable: if two curves describe adjacent pieces of the shape, they should connect smoothly.

We actually have two problems that we want to solve:

1. The exact curve is known, and we want to approximate it, for instance by something that is cheaper to compute, or
2. Only certain points are given, and we want to draw a smooth curve through them.

We will tackle the second problem first.

1.1 Interpolation

The interpolation problem is that of, given points $(x_1, f_1) \ldots (x_n, f_n)$, drawing a curve through them, for instance for purposes of computing intermediate values. Suppose that we have decided on a polynomial for the interpolating curve. With n points we need an

$n-1$st degree polynomial $p(x) = p_{n-1}x^{n-1} + \cdots + p_1x + p_0$, which takes n coefficients. We can draw up the set of equations $p(x_i) = f_i$ and solve that. The system

$$\begin{array}{rcl} p_{n-1}x_1^{n-1} + \cdots + p_1x_1 + p_0 &=& f_1 \\ &\cdots& \\ p_{n-1}x_n^{n-1} + \cdots + p_1x_n + p_0 &=& f_n \end{array}$$

can be written as $X\bar{p} = \bar{f}$, where

$$X = \left(x_i^j\right), \quad \bar{p} = \begin{pmatrix} p_1 \\ \vdots \\ p_{n-1} \end{pmatrix}, \quad \bar{f} = \begin{pmatrix} f_1 - p_0 \\ \vdots \\ f_n - p_0 \end{pmatrix}$$

Solving this system is not overly expensive, but its numerical stability is questionable. A better way of computing the same polynomial p is to define auxiliary polynomials $p^{(k)}$:

$$p^{(k)}(x) = c_k(x - x_1)\cdots(x - x_{k-1})\,(x - x_{k+1})\cdots(x - x_n)$$

where c_k is chosen so that $p^{(k)}(x_k) = 1$. From the fact that $p^{(i)}(x_j) = \delta_{ij}$, it follows that

$$p(x) = \sum_i f_i p^{(i)}(x), \qquad p^{(i)}(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} \tag{1}$$

interpolates the points as intended. It is easy enough to prove that polynomials are uniquely defined by these interpolation points, so we have now computed the same polynomial in a more stable way. A polynomial that is based on exact interpolation of values in a number of points is called a 'Lagrange interpolation' polynomial.

Another type of interpolation is 'Hermite interpolation', where the derivatives are dictated, rather than function values. Analogous to the above construction, we can define polynomials

$$q^{(k)} = c_k(x - x_1)^2 \cdots (x - x_{k-1})^2 \cdot (x - x_k) \cdot (x - x_{k+1})^2 \cdots (x - x_n)^2$$

where c_k is chosen so that $q^{(k)'}(x_k) = 1$.

1.2 Approximation

The above notion of interpolation is sometimes applied to known curves. For instance, by finding an interpolating polynomial we may aim to find a cheaper way of computing values on the curve. This raises the question how well the interpolation curve approximates the original curve.

In classical approximation theory there is usually a family of functions $\{f_n\}$, typically polynomials of increasing degree, such that $\|f_n - f\| \to 0$, typically on a closed interval I. The Weierstrass approximation theorem tells us that every continuous function on a closed bounded interval can be approximated by polynomials.

Note that this is uniform convergence:

$$\forall_\epsilon \exists_N \forall_{x \in I, n \geq N} : |f_n(x) - f(x)| \leq \epsilon.$$

This is a stronger statement than pointwise convergence:

$$\forall_{x \in I, \epsilon} \exists_N \forall_{n \geq N} : |f_n(x) - f(x)| \leq \epsilon.$$

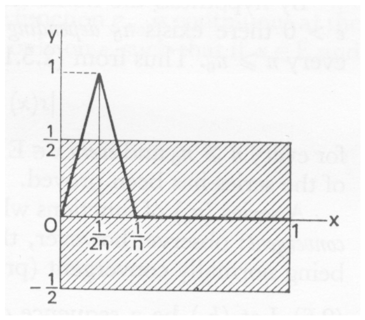


Figure 1: A family of functions that converges pointwise but not uniformly.

It is easy to find families of functions f_n that convergence in the pointwise sense, but not uniformly; see figure **??**.

The spline curves that we will study later in this chapter are a special case of Bernstein polymials: the n-th Bernstein polynomials for a function f is

$$B_n(f)(t) = \sum_{p=0}^{n} \binom{n}{p} f\left(\frac{p}{n}\right)(1-t)^{n-p}t^p.$$

If f is continuous on $[0,1]$, this sequence converges uniformly to f. It is worth remarking that these polynomials do not require computation of derivatives.

While the ideas behind Lagrange and Hermite interpolation will find applicability later in this story, the idea of interpolating with a single, high degree, polynomial may not be a good one from a point of uniform convergence. The error can be unacceptably large, as can be seen in figure **??**, where the dashed line is an interpolation on equispaced points. In this case there is in fact not even pointwise convergence. There are a few ways out of that, such as better choice of interpolation points or of basis functions. In figure **??** the dotted line uses Tchebyshev interpolation points which is seen to remedy the problem to a large extent.

However, the approach we will use here is that of piecewise approximations with relatively low degree polynomials. This simplifies certain aspects, but we need to take care to piece together such curves smoothly. For instance, with Lagrange interpolation the direction of the curve at the end points can not be specified.

1.3 Computation with interpolation curves

While we will mostly investigate theoretical properties of interpolation curves, the practical matter of how efficient it is to work with them, also deserves attention. In equation (**??**) there are n terms involving n multiplications and additions each, making for an $O(n^2)$ cost. This can be considerably reduced by rewriting the formula as

$$p(x) = \prod_i (x - t_i) \cdot \sum_i \frac{y_i}{x - t_i}, \qquad y_i = f_i / \prod_{j \neq i} (x_i - x_j),$$

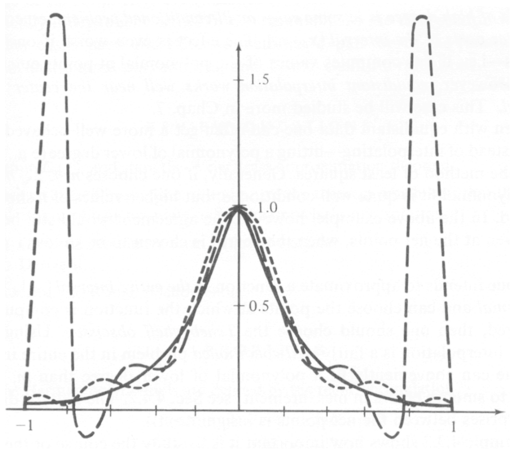


Figure 2: The Runge effect of interpolating with a high degree polynomial

which takes n additions, multiplications, and additions if the y_i quantities are precomputed. We will now see a formulation that dispenses with the divisions, and that will also be simpler if derivatives need to be computed.

The k-th 'divided difference' of a function g in the points $\tau_1 \ldots \tau_{k+1}$, notation $[\tau_1, \ldots, \tau_{k+1}]g$, is the leading coefficient of the k-th order[1] polynomial p_{k+1} that agrees with g in the points $\tau_1 \ldots \tau_{k+1}$.

The simplest examples of divided differences are:

- The zeroeth divided difference of a function is the leading coefficient of a zeroth order polynomial $p(x) = c$ that agrees with that function in one point: $g(\tau_1) = g_1$. Clearly $[\tau_1]g = g_1$.
- The first divided difference is the leading coefficient of a linear function that agrees in two points:
$$[\tau_1, \tau_2]g = \frac{g(\tau_2) - g(\tau_1)}{\tau_2 - \tau_1} = \frac{[\tau_2]g - [\tau_1]g}{\tau_2 - \tau_1}$$
This equation may suggest to the reader a relation for higher divided differences. We shall see in lemma **??** that this indeed holds.

We now prove some facts about divided differences.

Lemma 1 *Let $p_{k+1} \in \prod_{<k+1}$ agree with g in $\tau_1 \ldots \tau_{k+1}$, and $p_k \in \prod_{<k}$ with g in $\tau_1 \ldots \tau_k$, then*

$$p_{k+1}(x) - p_k(x) = [\tau_1, \ldots, \tau_{k+1}]g \prod_{i=1}^{k} (x - \tau_i). \tag{2}$$

1. It is convenient to talk about polymomials of a certain order rather than degree: a polynomial of order k has a degree no more than k. We denote this set with $\prod_{<k+1}$. One advantage of this set is that it is closed under summing.

Proof. Since p_k is of a lower order, we immediately have

$$p_{k+1} - p_k = [\tau_1, \ldots, \tau_{k+1}]gx^k + cx^{k-1} + \cdots.$$

Observing that $p_{k+1} - p_k$ is zero in t_i for $i \leq k$, it follows that

$$p_{k+1} - p_k = C \prod_{i=1}^{k} (x - \tau_i).$$

From this we get that $C = [\tau_1, \ldots, \tau_{k+1}]g$. •

If we repeat this lemma we find that

$$p_{k+1}(x) = \sum_{m=1}^{k+1} [\tau_1, \ldots, \tau_m]g \prod_{i=1}^{m-1} (x - \tau_i), \tag{3}$$

which we can evaluate as

$$\begin{array}{rcl} p_{k+1}(x) & = & [\tau_1, \ldots, \tau_{k+1}]g \prod^k (x - \tau_i) + [\tau_1, \ldots, \tau_k]g \prod^{k-1} (x - \tau_i) \\ & = & [\tau_1, \ldots, \tau_{k+1}]g(x - \tau_k)\big(c_k + [\tau_1, \ldots, \tau_k]g(x - \tau_{k-1})\big(c_{k-1} + \cdots \end{array}$$

where $c_k = [\tau_1, \ldots, \tau_k]g/[\tau_1, \ldots, \tau_{k+1}]g$. This is a very efficient evaluation by Horner's rule.

The remaining question is how to construct the divided differences. We approach this recursively.

Lemma 2 *Divided differences can be constructed by, eh, dividing differences*

$$[\tau_1, \ldots, \tau_{n+1}]g = ([\tau_1, \ldots, \tau_n]g - [\tau_2, \ldots, \tau_{n+1}]g) / (\tau_1 - \tau_{n+1}). \tag{4}$$

Proof. Let three polynomials be given:

- $p_n^{(1)} \in \prod_{<n}$ agrees with g on $\tau_1 \ldots \tau_n$;
- $p_n^{(2)} \in \prod_{<n}$ agrees with g on $\tau_2 \ldots \tau_{n+1}$;
- $p_{n-1} \in \prod_{<n-1}$ agrees with g on $\tau_2 \ldots \tau_n$.

Then by lemma **??**

$$\begin{array}{rcl} p_n^{(1)} - p_{n-1} & = & [\tau_1, \ldots, \tau_n]g \prod_{j=2}^{n} (x - \tau_j) \\ p_n^{(2)} - p_{n-1} & = & [\tau_2, \ldots, \tau_{n+1}]g \prod_{j=2}^{n} (x - \tau_j) \end{array}$$

Now let p_{n+1} be the polynomial that agrees with g on $\tau_1 \ldots \tau_{n+1}$, then

$$\begin{array}{rcl} p_{n+1} - p^{(1)} & = & [\tau_1, \ldots, \tau_{n+1}]g \prod_{j=1}^{n} (x - \tau_j) \\ p_{n+1} - p^{(2)} & = & [\tau_1, \ldots, \tau_{n+1}]g \prod_{j=2}^{n+1} (x - \tau_j) \end{array}$$

Subtracting both pairs of equations, we find two expressions for $p_n^{(1)} - p_n^{(2)}$:

$$([\tau_1, \ldots, \tau_n]g - [\tau_2, \ldots, \tau_{n+1}]g) \prod_{j=2}^{n} (x - \tau_j) = [\tau_1, \ldots, \tau_{n+1}]g \left(\prod_{j=2}^{n+1} - \prod_{j=1}^{n} \right) (x - \tau_j)$$

Filling in $\tau_2 \ldots \tau_n$ in this equation, we find zero for both sides. Using $x = \tau_1$ we find

$$([\tau_1, \ldots, \tau_n]g - [\tau_2, \ldots, \tau_{n+1}]g) \prod_{j=2}^{n} (\tau_1 - \tau_j) = [\tau_1, \ldots, \tau_{n+1}]g \prod_{j=2}^{n+1} (\tau_1 - \tau_j)$$

from which the lemma follows. •

From this lemma, we see easily that $[\tau_1, \dots, \tau_n]g$ can be computed in approximately $n^2/2$ additions and divisions.

2 Parametric curves

So far we have been looking at approximation by a function of a single value. That is, we have y as a function of x. This rules out many curves, such as circles. We could try expressing x as a function of y, or more generally rotating them, but that is a big hassle, and it would still rule out some curves.

Another approach is to let the curve be defined implicitly by $f(x, y, z) = 0$. This suffers from several problems. We could get too many solutions, for instance as in $x^2 + y^2 - 1 = 0$, requiring constraints. Tracking a path is possible in theory, but is not trivial in practice. Finally, many shapes are defined piecewise, and joining shapes in a smooth way is also tricky.

Rather than considering a curve in the plane as a function, it is more productive to describe the shape of the curve, as some imaginary point moves over the curve. That is, we have a description of the points on the curve as

$$P = P(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}.$$

(Feel free to think of t as time.)

A simple example of how this works is to consider two points P_1 and P_2, and the curve $P = tP_2 + (1-t)P_1$. Then for $t = 0$, $P(0) = P_1$, for $t = 1$, $P(1) = P_2$, and for intermediate values of t we get some point between P_1 and P_2.

That this solves our problems with multiple solutions that were present in both function and implicit approaches is clear if we look at the example $P(t) = (\cos 2\pi t, \sin 2\pi t)$, which traces out a circle as t goes from 0 to 1.

While a description in terms of piecewise linear functions would be very simple, it is not smooth. Using the various ideas sketched above, we will now concentrate on curves that are piecewise parametric cubic splines. Cubics have the following property that they are the lowest degree that allows specification of location and direction in the end points. Higher degree functions would allow for instance the specification of higher derivatives, but unless great care is taken, they would introduce unwanted ‘wiggles’ in the curves.

Using piecewise cubic parametric curves then is a good mean between ease of use and power of approximation.

2.1 Parametrized basis functions

To begin with we will concentrate on a single curve $Q(t)$ where $t = 0 \ldots 1$. We often write this as $Q(t) = C \cdot T$ where the coefficient matrix

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \end{pmatrix}, \qquad T = \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

The direction of the curve is then given by

$$\frac{dQ(t)}{dt} = C \cdot \frac{dT}{dt} = C \cdot \begin{pmatrix} 0 \\ 1 \\ 2t \\ 3t^2 \end{pmatrix}$$

We see that the C matrix can be derived if we know a total of four locations or directions of the curve. For instance, if $P_1 = Q(0)$, $R_1 = Q'(0)$, $P_2 = Q(1)$, and $R_2 = Q'(1)$ are given, then

$$C \cdot \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 3 \end{pmatrix} = [P_1, R_1, P_2, R_2],$$

from which C follows.

Now, often we have a set of basis polynomials given, and we want to take combinations of them to satisfy the constraints. That can be done by splitting $C = GM$, where M describes the basis polynomials, and G is the 'geometry matrix'. We get the equation

$$Q(t) = G \cdot M \cdot T = \begin{pmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \end{pmatrix} \cdot \begin{pmatrix} m_{11} & \ldots & m_{14} \\ \vdots & & \vdots \\ m_{41} & \ldots & m_{44} \end{pmatrix} \cdot T \tag{5}$$

If we introduce new basis polynomials $\pi_i(t) = M_{i*} \cdot T$, then we see that $Q_x = G_{11}\pi_1 + G_{12}\pi_2 + G_{13}\pi_3 + G_{14}\pi_4$, $Q_y = G_{21}\pi_1 + \cdots$, et cetera.

2.2 Hermite basis functions

In equation (**??**) the matrix M describes the basis functions, so it is fixed for a certain class of curves: we will have one set of basis functions for Lagrange type curves, one for Hermite curves, et cetera. However, we have not yet seen a way to compute the matrix M.

The geometry matrix G is used to derive a specific curve in the class described by M: each choice of G corresponds to one curve. The columns of G will be points or direction vectors that somehow describe the intended curve.

Let us now consider Hermite curves. Here we want G to be the matrix of geometric constraints, $[P_1, R_1, P_2, R_2]$ in the above example. Recall that these constraints, using the locations of the end points and the derivatives of the curve there, give us indeed an example of Hermite interpolation.

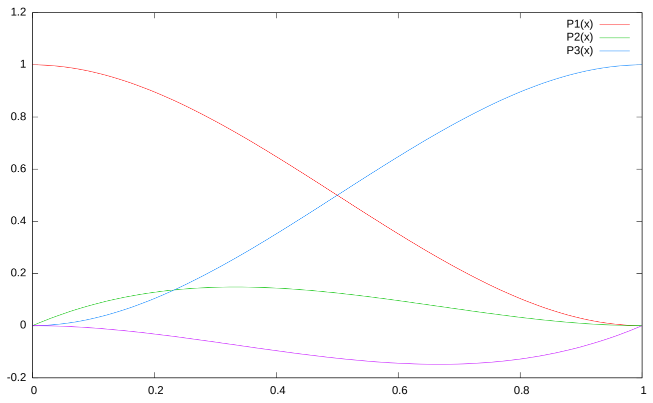


Figure 3: Hermite polynomials

We write out the equations. From $Q = G \cdot M \cdot T$ we get

$$Q(t) = G_H \cdot M_H \cdot T(t), \qquad Q'(t) = G_H \cdot M_H \cdot T'(t).$$

Applying both these formulas to $t = 0$ and $t = 1$, we get

$$Q_H \equiv [Q(0), Q'(0), Q(1), Q'(1)] = G_H \cdot M_H \cdot T_H$$

where

$$T_H = [T(0), T'(0), T(1), T'(1)] = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 3 \end{pmatrix}$$

But this Q_H is the matrix that we had stipulated as the matrix of geometry constraints, in other words: $G_H = Q_H$. It now follows that

$$M_H = T_H^{-1} = \begin{pmatrix} 1 & 0 & -3 & 2 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 3 & -2 \\ 0 & 0 & -1 & 1 \end{pmatrix}.$$

Writing this out, we find the cubic Hermite polynomials

$$P_1(t) = 2t^3 - 3t^2 + 1, \quad P_2(t) = t^3 - 2t^2 + t, \quad P_3(t) = -2t^3 + 3t^2, \quad P_1(t) = t^3 - t^2$$

illustrated in figure **??**, and we get $Q(t) = G \cdot B_H$ where $B_H = M \cdot T$ is the matrix of 'Hermite blending functions'.

```
#
# 4 cubic Hermite polynomials
#
set terminal pdf
set xrange [0:1]
set yrange [-.2:1.2]
P1(x) = 2*x**3-3*x**2+1
P2(x) = x**3-2*x**2+x
P3(x) = -2*x**3+3*x**2
P4(x) = x**3-x**2
plot P1(x), P2(x), P3(x), P4(x) title ""
```

Figure 4: The `gnuplot` source for figure **??**

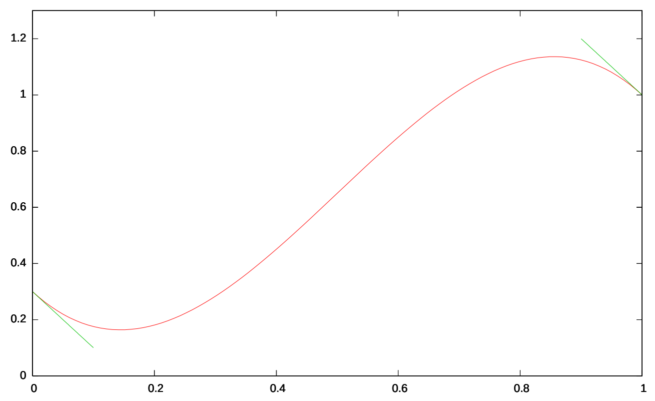


Figure 5: An example of Hermite interpolation

With this formula, we can take a set of geometric constraints, in this case the two endpoints and the directions there, and derive the formula for the Hermite curve that satisfies these constraints. As an example, figure **??** is the curve $.3P_1 - 2P_2 + P_3 - 2P4,$ that is, the curve through $(0, .3)$ and $(1, 1)$, with slope -2 in both $x = 0, 1$.

We make the transition to parametric curves by expressing both components as Hermite curves. For instance, figure **??** shows the curve

$$Q_x(t) = .1 * P_1(t) + P_2(t) + .9 * P_3(t), \quad Q_y(t) = .2 * P_1(t) + .3 * P_3(t) - P_4(t)$$

that is

$$Q = \begin{pmatrix} .1 \\ .2 \end{pmatrix} P_1 + \begin{pmatrix} 1 \\ 0 \end{pmatrix} P_2 + \begin{pmatrix} .9 \\ .3 \end{pmatrix} P_3 + \begin{pmatrix} 0 \\ -1 \end{pmatrix} P_4.$$

There is one aspect of these Hermite curves worth remarking on. In figure **??** we have replaced the direction vector $\binom{1}{0}$ in $(0, 0)$ by $\binom{x}{0}$, where $x = 1, 1.5, 2, 2.5,$ which all have the same direction, but a different magnitude. Clearly there is a visual effect.

```
#
# Hermite interpolation
#
set terminal pdf
set multiplot
set xrange [0:1]
set yrange [0:1.3]
P1(x) = 2*x**3-3*x**2+1
P2(x) = x**3-2*x**2+x
P3(x) = -2*x**3+3*x**2
P4(x) = x**3-x**2
```

```
p1y = .3
p1slope = -2
p2y = 1
p2slope = -2
plot p1y*P1(x) + p1slope*P2(x) \
  + P3(x) + p2slope*P4(x) title ""
set parametric
set style function lines
plot [t=0:.1] t,  p1y+t*p1slope \
  title "" with lines 2
plot [t=0:.1] 1-t,p2y-t*p2slope \
  title "" with lines 2
```

Figure 6: The source for figure **??**

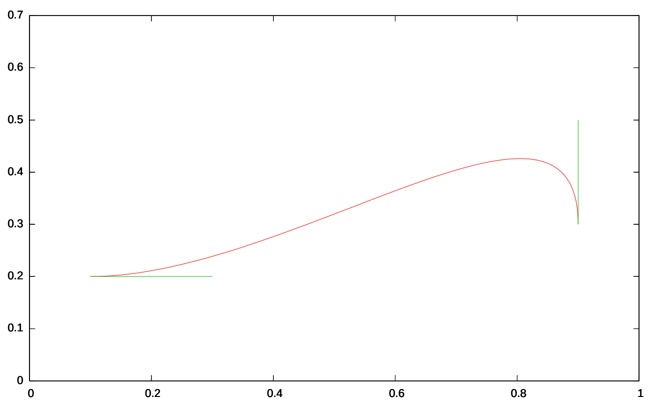


Figure 7: A Hermite interpolation curve

2.3 Splines

We will now take a close look at Bernshtein polynomials of degree 3:

$$z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2 + 3(1-t)t^2 z_3 + t^3 z_4, \tag{6}$$

also known as Bezier curves or Bezier cubics after Pierre Bezier, an engineer at Renault[2].

There are a couple of ways of looking at these polynomials. Consider the function $z(t)$ to be the sum of four basis functions, $(1-t)^3$, $(1-t)^2 t$, $(1-t)t^2$, and t^3, each multiplied by a factor deriving from a control point. From the formulas, and from a picture (figure **??**) we see that the first term $p_1(t) = (1-t)^3$ is the only one with $p(0) \neq 0$. Likewise, p_4 is the only one with $p(1) \neq 0$. That means $z(0) = z_1$ and $z(1) = z_4$. Furthermore, the second term is (after the first term) the only remaining one with $p'(0) \neq 0$, so by choosing z_2 we can change $z'(0)$ without changing $z(0)$ or $z(1)$. Likewise z_3 for $z'(1)$.

2. Pierre Bézier was born September 1, 1910 and died November 25, 1999 at the age of 89. In 1985 he was recognized by ACM SIGGRAPH with a 'Steven A. Coons' award for his lifetime contribution to computer graphics and interactive techniques.

```
#
# Parametric Hermite curve
#
set terminal pdf
set parametric
set multiplot
set xrange [0:1]
set yrange [0:.7]
P1(t) = 2*t**3-3*t**2+1
P2(t) = t**3-2*t**2+t
P3(t) = -2*t**3+3*t**2
P4(t) = t**3-t**2
```

```
p1x  = .1 ; p1y  = .2
p1dx =  1 ; p1dy =  0
p2x  = .9 ; p2y  = .3
p2dx =  0 ; p2dy = -1
plot [t=0:1] \
  p1x*P1(t)+p1dx*P2(t)+p2x*P3(t)+p2dx*P4(t), \
  p1y*P1(t)+p1dy*P2(t)+p2y*P3(t)+p2dy*P4(t) \
  title ""
plot [t=0:.2] p1x+t*p1dx,p1y+t*p1dy \
  title "" with lines 2
plot [t=0:.2] p2x-t*p2dx,p2y-t*p2dy \
  title "" with lines 2
```

Figure 8: The source of figure **??**

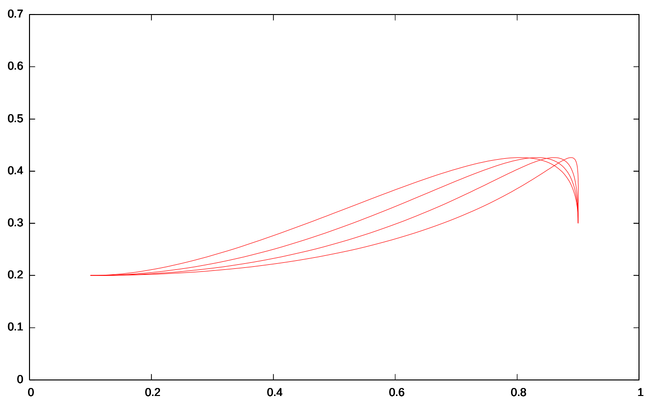


Figure 9: The same curve, tinkering with the direction vector

Bezier curves can be derived from cubic Hermite splines by replacing the direction vectors R_1, R_2 by control points P_1', P_2', so that $R_1 = 3(P_1' - P1)$ and $R_2 = 3(P_2 - P_2')$. For the Bezier geometry vector we then have

$$G_B = [P_1, P_1', P_2', P_2]$$

and the relation with the Hermite geometry vector

$$G_H = [P_1, R_1, P_2, R_2] = [P_1, P_1', P_2', P_2]M_{BH} = G_B \cdot M_{BH}$$

```
set terminal pdf
set parametric
set multiplot
set dummy t
set xrange [0:1]
set yrange [0:.7]
P1(t) = 2*t**3-3*t**2+1
P2(t) = t**3-2*t**2+t
P3(t) = -2*t**3+3*t**2
P4(t) = t**3-t**2
p1x = .1  ; p1y = .2
p2x  = .9 ; p2y = .3
p2dx = 0  ; p2dy = -1
# direction 1:
p1dx = 1 ; p1dy = 0
plot [t=0:1] \
  p1x*P1(t)+p1dx*P2(t)+p2x*P3(t)+p2dx*P4(t), \
  p1y*P1(t)+p1dy*P2(t)+p2y*P3(t)+p2dy*P4(t) \
  title ""
```

```
# direction 2:
p1dx = 1.5 ; p1dy = 0
plot [t=0:1] \
  p1x*P1(t)+p1dx*P2(t)+p2x*P3(t)+p2dx*P4(t), \
  p1y*P1(t)+p1dy*P2(t)+p2y*P3(t)+p2dy*P4(t) \
  title ""
# direction 3:
p1dx = 2 ; p1dy = 0
plot [t=0:1] \
  p1x*P1(t)+p1dx*P2(t)+p2x*P3(t)+p2dx*P4(t), \
  p1y*P1(t)+p1dy*P2(t)+p2y*P3(t)+p2dy*P4(t) \
  title ""
# direction 4:
p1dx = 2.5 ; p1dy = 0
plot [t=0:1] \
  p1x*P1(t)+p1dx*P2(t)+p2x*P3(t)+p2dx*P4(t), \
  p1y*P1(t)+p1dy*P2(t)+p2y*P3(t)+p2dy*P4(t) \
  title ""
```

Figure 10: The source for figure **??**

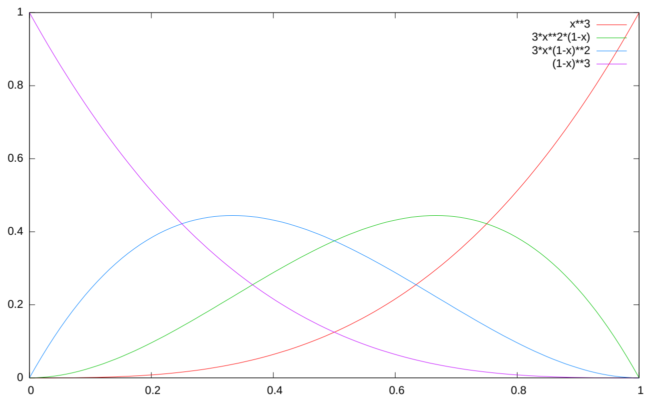


Figure 11: Bernshtein polynomials

where

$$M_{BH}=\begin{pmatrix}1&-3&0&0\\0&3&0&0\\0&0&0&-3\\0&0&1&3\end{pmatrix} \tag{7}$$

Defining

$$M_B=M_{BH}\cdot M_H=\begin{pmatrix}1&-3&3&-1\\0&3&-6&3\\0&0&3&-3\\0&0&0&1\end{pmatrix} \tag{8}$$

we now get for Bezier curves

$$Q(t)=G_H\cdot M_H\cdot T(t)=G_B\cdot M_{BH}\cdot M_H\cdot T(t)=G_B\cdot M_B\cdot T(t)$$

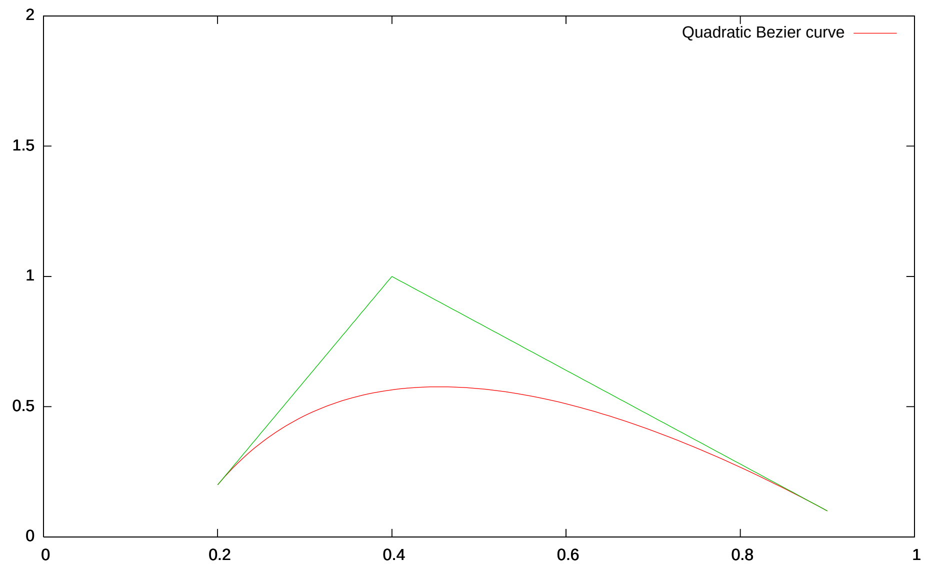
We can also write that as $Q_x(t)=g_{11}B_1(t)+g_{12}B_2(t)+\cdots$ where

$$\begin{array}{lll}B_1(t) & =1-3t+3t^2-t^3 & =(1-t)^3\\ B_2(t) & =3t-6t^2+3t^3 & =3t(1-t)^2\\ B_3(t) & =3t^2-3t^3 & =3t^2(1-t)\\ B_4(t) & = & =t^3\end{array}$$

which are the Bernstein polynomials we started this section with.

The sum of these polynomials (this is equivalent to setting the z_i coefficients to one in equation (**??**)) is $z(t)=(t+(1-t)))^3\equiv 1$. Also, the polynomials are positive on $[0,1]$, so the components Q_x,Q_y,Q_z are weighted averages of the polynomials. This means that the curve, consisting of weighted sums of the control points, is contained in the convex hull of the control points.

Exercise 1. One could also define quadratic Bezier curves. These have only a single control point, and the curve is such that in both the endpoints it is aimed at the control point.



Derive the basis functions and geometry matrix for this case. Make a `gnuplot` figure of a single quadratic Bezier curve, and of two curves joined smoothly. Hint: you can follow the construction of the cubic splines in the lecture notes. The only problem is defining the control point. First draw up the Hermite geometry matrix based on end points q_0 and q_1, and the derivative q_0' in the first end point. Derive from them the derivative q_1' in the other end point. The control point then lies on the intersection of two lines. Solving this looks like a single equation in two unknowns, but it can be solved: write it as a matrix-vector equation that is satisfied no matter the choice of the geometry matrix.

2.4 Calculation of Bezier curves

Suppose we have a Bezier curve defined by four control points, and we want to draw points on the curve, meaning that we have to evaluate the function $Q(t)$ for a number of values of t. The relation $Q(t) = G \cdot M \cdot T(t)$ allows us to do this calculation in

- 2 multiplications to form the terms t^2 and t^3 in T;
- 16 multiplications and 12 additions forming $M \cdot T$;
- 12 multiplications and 9 additions forming $G \cdot (M \cdot T)$.

An obvious improvement is to store $\tilde{M} = G \cdot M$, which brings the cost down to two multiplications for T and

- 12 multiplications and 9 additions for forming $\tilde{M} \cdot T$.

A similar count is arrived at by looking at divided differences. Additionally, this way of computing is more stable.

From the formula $Q(t) = G \cdot M \cdot T(t)$ we get for each component

$$q_i(t) = \sum_j G_{ij}(MT)_j = \sum_{j,k} G_{ij} M_{jk} t^{k-1}.$$

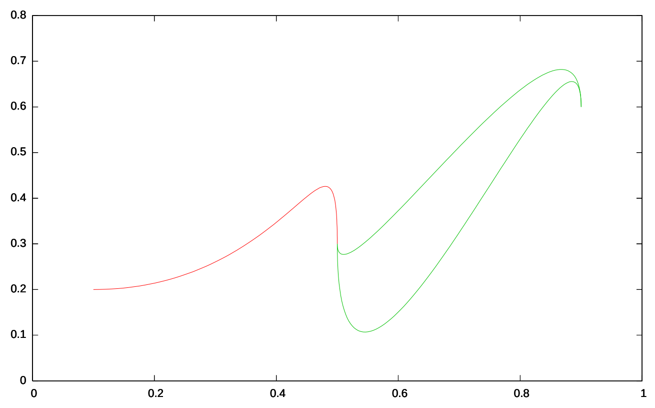


Figure 12: Two Hermite curves, joined together smoothly in $(.5, .3)$

Looking at only one component for simplicity, we find, for instance

$$x(t) = \sum_k c_k t^{k-1} \qquad c_k = \sum_j G_{1j} M_{jk}.$$

We recall equation (**??**):

$$M_B = \begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and writing $g_j \equiv G_{1j}$ we find

$$c_1 = g_1, \quad c_2 = 3(g_2 - g_1), \quad c_3 = 3(g_3 - 2g_2 + g_1), \quad c_4 = g_4 - 3g_3 + 3g_2 - g_1.$$

In this we recognize divided differences of g:

$$\begin{array}{lcl} [2,1]g & = & g_2 - g_1, \\ [3,2,1]g & = & [3,2]g - [2,1]g = (g_3 - g_2) - (g_2 - g_1) \\ & = & g_3 - 2g_2 + g_1 \\ [4,3,2,1] & = & [4,3,2]g - [3,2,1]g = (g_4 - 2g_3 + g_2) - (g_3 - 2g_2 + g_1) \\ & = & g_4 - 3g_3 + 3g_2 - g_1 \end{array}$$

using lemma **??**.

3 Practical use

3.1 Piecewise curves

As we indicated earlier, complicated shapes can be approximated by piecewise cubics. Figure **??** shows two Hermite curves joined together. The curve is continuous and the directions at the join are proportional. This is called ‘geometric continuity’, and is denoted G^1.

So-called B-splines (‘basis splines’) are constructed by piecing together Bezier curves, not only continuously, but differentiably. For this, if $P_1 \dots P_4$ and $P_4 \dots P_7$ are the control points of the two curves, then we require

$$P_4 = (P_3 + P_5)/2.$$

This is shown in figure **??**.

3.2 Drawing splines

Even if we can evaluate a Bezier curve efficiently (section **??**) in a given point, we can improve on this considerably in the context of line drawing. Consider the problem of drawing a cubic curve on the interval $[0, 1]$ by drawing consecutive points $Q(n\delta)$ for $n = 0, 1, \dots$.

We will discuss one line-drawing technique in the chapter on raster graphics. A technique that is attractive for splines, and which is used in METAFONT, is recursive subdivision. Here, a curve is subdivided until a segment is a straight line within the pixel resolution, so that a line drawing algorithm can be used. The test for a straight line can be implemented efficiently through using the spline control points.

```
set terminal pdf
set parametric
set multiplot
set dummy t
set xrange [0:1]
set yrange [0:.8]
P1(t) = 2*t**3-3*t**2+1
P2(t) = t**3-2*t**2+t
P3(t) = -2*t**3+3*t**2
P4(t) = t**3-t**2
p1x = .1 ; p1y = .2
p2x =  1 ; p2y =  0
p3x = .5 ; p3y = .3
p4x =  0 ; p4y = -1
plot [t=0:1] \
  p1x*P1(t)+p2x*P2(t)+p3x*P3(t)+p4x*P4(t), \
  p1y*P1(t)+p2y*P2(t)+p3y*P3(t)+p4y*P4(t) \
  title ""
p5x = .9 ; p5y = .6
p6x =  0 ; p6y = -1
plot [t=0:1] \
  p3x*P1(t)+.5*p4x*P2(t)+p5x*P3(t)+p6x*P4(t), \
  p3y*P1(t)+.5*p4y*P2(t)+p5y*P3(t)+p6y*P4(t) \
  title "" with lines 2
plot [t=0:1] \
  p3x*P1(t)+2*p4x*P2(t)+p5x*P3(t)+p6x*P4(t), \
  p3y*P1(t)+2*p4y*P2(t)+p5y*P3(t)+p6y*P4(t) \
  title "" with lines 2
```

Figure 13: The source for figure **??**

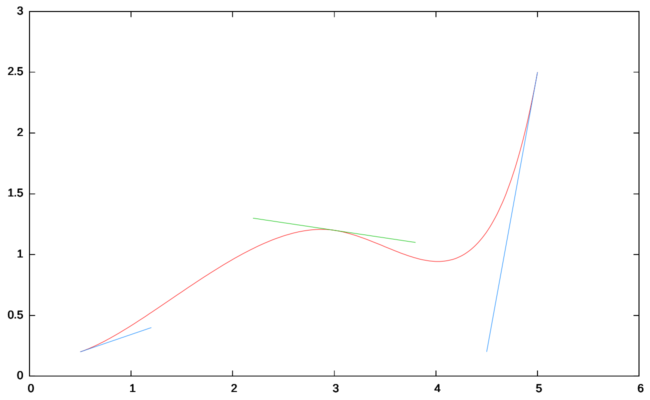


Figure 14: Two Bezier curves, joined together smoothly

```
set terminal pdf
set xrange [0:6]
set yrange [0:3]
set parametric
B1(x) = x**3
B2(x) = 3*x**2*(1-x)
B3(x) = 3*x*(1-x)**2
B4(x) = (1-x)**3
P1x = .5  ; P1y = .2
P2x = 1.2 ; P2y = .4
P3x = 2.2 ; P3y = 1.3
P4x =   3 ; P4y = 1.2
P5x = 2*P4x-P3x
P5y = 2*P4y-P3y
P6x = 4.5 ; P6y = .2
P7x = 5   ; P7y = 2.5
set multiplot
```

```
plot [t=0:1] \
  P1x*B1(t)+P2x*B2(t)+P3x*B3(t)+P4x*B4(t), \
  P1y*B1(t)+P2y*B2(t)+P3y*B3(t)+P4y*B4(t) \
  title ""
plot [t=0:1] \
  P4x*B1(t)+P5x*B2(t)+P6x*B3(t)+P7x*B4(t), \
  P4y*B1(t)+P5y*B2(t)+P6y*B3(t)+P7y*B4(t) \
  title ""
plot [t=-1:1] \
  P4x+t*(P5x-P4x),P4y+t*(P5y-P4y) \
  title "" with lines 2
plot [t=0:1] \
  P1x+t*(P2x-P1x),P1y+t*(P2y-P1y) \
  title "" with lines 3
plot [t=0:1] \
  P7x+t*(P6x-P7x),P7y+t*(P6y-P7y) \
  title "" with lines 3
```

Figure 15: The source for figure **??**

Contents