3.4 The Unit Interval

We want to build a basis for each subinterval of our spline (and later figure out how to connect them together to get continuity and smoothness). It's a bit messy, however, constructing a nice basis for a general interval $[t_i, t_{i+1}]$. However, we can easily transform this interval to and from the unit interval $[0, 1]$, using a change of variable:

$$\begin{aligned} t = (t_{i+1} - t_i)s + t_i \qquad &\Leftrightarrow \qquad s = \frac{t - t_i}{t_{i+1} - t_i} \\ t \in [t_i,\, t_{i+1}] \qquad &\Leftrightarrow \qquad s \in [0,\, 1] \end{aligned} \tag{17}$$

Then if we build a basis for polynomials on the unit interval, say $\{\phi_i(s)\}$, we can immediately convert it into a basis for the spline interval plugging in the subsitution for s in terms of t.

Note that this transformation is itself a linear polynomial. You can view t (as a function of s) as a lerp between t_i at $s = 0$ and t_{i+1} at $s = 1$. Alternatively, s (as a function of t) is a lerp between 0 at $t = t_i$ and 1 at $t = t_{i+1}$; in fact, its formula already showed up in the lerping we spelled out above.

3.5 Hermite Splines

Our main motivation for going to higher degree polynomials was the pursuit of smoothness. In particular, to achieve a C^1 spline, we need to make sure that not only does the value of the function match from both sides of a knot, but also its derivative.

To get continuity, C^0, we made the function match by requiring it to be interpolating: we actually had the user provide the control point values $\{f_i\}$ and constrained the polynomials to equal the control point values at the knots.

We can take the same idea for achieving C^1: ask the user to provide values for the function's derivative at the knots as well, and constrain the polynomials to have a matching first derivative at the knots. This is what is called a *Hermite spline*, where derivative data is supplied and is interpolated.

Let's work this out for the unit interval. Say we want a polynomial on $[0, 1]$ where the values interpolate f_0 and f_1,

$$f(0) = f_0 \qquad f(1) = f_1, \tag{18}$$

and the derivatives (or slopes) interpolate g_0 and g_1,

$$f'(0) = g_0 \qquad f'(1) = g_1. \tag{19}$$

This is four equations, for which we will want four unknown coefficients to solve—i.e. we will want a cubic polynomial.

For now let's work with the standard basis, $\{1, s, s^2, s^3\}$. A generic cubic is $f(s) = a_0 + a_1 s + a_2 s^2 + a_3 s^3$, with derivative $f'(s) = a_1 + 2a_2 s + 3a_3 s^2$. Substituting this into our constraints gives:

$$\begin{aligned} f(0) &= a_0 &&= f_0 \\ f(1) &= a_0 + a_1 + a_2 + a_3 &&= f_1 \\ f'(0) &= a_1 &&= g_0 \\ f'(1) &= a_1 + 2a_2 + 3a_3 &&= g_1 \end{aligned} \tag{20}$$

Solving this system (with a little effort) gives:

$$\begin{aligned} a_0 &= f_0 \\ a_1 &= g_0 \\ a_2 &= 3f_1 - 3f_0 - 2g_0 - g_1 \\ a_3 &= 2f_0 - 2f_1 + g_0 + g_1 \\ \Rightarrow \quad f(s) &= f_0 + g_0 s + (3f_1 - 3f_0 - 2g_0 - g_1)s^2 + (2f_0 - 2f_1 + g_0 + g_1)s^3. \end{aligned} \tag{21}$$

We can rearrange this around the user data:

$$f(s) = f_0\left[1 - 3s^2 + 2s^3\right] + f_1\left[3s^2 - 2s^3\right] + g_0\left[s - 2s^2 + s^3\right] + g_1\left[-s^2 + s^3\right]. \tag{22}$$

Of course, what we see in square brackets here is a basis for cubic polynomials which naturally fits Hermite data on $[0,1]$. It's worth plotting these to get an idea for how they look, and in particular figuring out what they are equal to at the endpoints $s=0$ and $s=1$, as well as their slopes there.

There's a fly in the ointment, however. When we transform from $[0,1]$ to an interval $[t_i, t_{i+1}]$ of non-unit length, the derivative of the function is scaled (via the chain rule) by a factor of $1/(t_{i+1}-t_i)$. We need to scale the last two functions to make up for this. Likewise, if the user changes the length of the interval, our software would need to automatically rescale the slopes to match.

Specifying slopes themselves is also a bit less intuitive than function values for users. One approach is taken by *Catmull-Rom* splines, where the slopes are automatically computed from the control points themselves. We do this using finite differences, a numerical method for estimating the derivative of a function from function values, based on Taylor series. Without going into too many details, if control points (t_{i-1}, f_{i-1}), (t_i, f_i), and (t_{i+1}, f_{i+1}) are given, a good estimate of the derivative at t_i is:

$$f'(t_i) \approx g_i = \left(\frac{t_i - t_{i-1}}{t_{i+1}-t_{i-1}}\right)\left(\frac{f_{i+1}-f_i}{t_{i+1}-t_i}\right) + \left(\frac{t_{i+1}-t_i}{t_{i+1}-t_{i-1}}\right)\left(\frac{f_i - f_{i-1}}{t_i - t_{i-1}}\right) \tag{23}$$

This simplifies when the knots are uniformly spaced to

$$g_i = \frac{f_{i+1}-f_{i-1}}{t_{i+1}-t_{i-1}} \tag{24}$$

which is when Catmull-Rom is most commonly used. At the endpoints of the spline (where t_{-1} or t_{n+1} don't exist) an alternate estimate must be used, such as the simpler one-sided finite difference $g_0 = (f_1 - f_0)/(t_1 - t_0)$.

Catmull-Rom isn't always useful however. Sometimes its slope estimates aren't quite what the user wants, and it also enforces C^1 smoothness which might be undesirable in certain cases like collisions. With the Hermite version, a user can elect to have different slopes at a control point for an interval, deliberately *breaking* smoothness to model an impact. This will also be the case for our next spline.

Finally, for the mathematician inside all of us, the Hermite formulation (with or without Catmull-Rom slope estimates) lacks an elegant generalization to other degree polynomials such as quadratics, which is mathematically annoying.

For all these reasons, we need to go one step further into spline methods to get to the Bernstein polynomial basis used for Bézier curves.

3.6 The Cubic Bernstein Basis

One of the good things about the cubic Hermite basis is that it provides separate control of different aspects of the function: the coefficients actually have a direct intuitive meaning as the values or slopes at the endpoints. The explicit control of the slopes at the endpoints also makes it very convenient to achieve a C^1 function by matching slopes from both sides at a knot. We want to keep these features. However, the problems we identified really revolved around the fact that we had too much direct control of the slopes: two of the coefficients had a different meaning (value of derivative) than the "normal" (value of function).

If we want a polynomial on $s \in [0,1]$ to be one at $s = 0$, obviously its constant term has to be one. If we want it be zero there, it must have zero constant term or in other words have a factor of s; if we further want its derivative to also be zero at $s = 0$ it can't have a linear term and then must have a factor of s^2. The same analysis applies to the other endpoint, $s = 1$, only with factors of $1 - s$ instead. This suggests we should look at building basis functions from products of s and $(1 - s)$.

Building from this observation, the *Bernstein basis* for cubics can be inspired by looking at the following expansion:

$$\begin{aligned} 1 = 1^3 &= [(1-s)+s]^3 \\ &= (1-s)^3 + 3(1-s)^2 s + 3(1-s)s^2 + s^3. \end{aligned} \tag{25}$$

The individual terms are the basis functions we are looking for:

$$\begin{aligned}
B_{0,3} &= (1-s)^3, \\
B_{1,3} &= 3(1-s)^2 s, \\
B_{2,3} &= 3(1-s)s^2, \\
B_{3,3} &= s^3.
\end{aligned} \tag{26}$$

These are also worth plotting. Observe that $B_{0,3}(0) = 1$ whereas all the others evaluate to zero there, and likewise for $B_{3,3}(1)$. This means the coefficients for $B_{0,3}$ and $B_{3,3}$ will be the function values at the endpoints which we want to interpolate.

Consider a generic cubic "Bernstein polynomial", i.e. a linear combination of the Bernstein basis functions:

$$f(s) = aB_{0,3}(s) + bB_{1,3}(s) + cB_{2,3}(s) + dB_{3,3}(s). \tag{27}$$

Its derivative is:

$$\begin{aligned}
f'(s) = aB'_{0,3}(s) + bB'_{1,3}(s) + cB'_{2,3}(s) + dB'_{3,3}(s) \\
= a[-3(1-s)^2] \;+\; b[-6(1-s)s + 3(1-s)^2] \\
+\, c[-3s^2] \;+\; 6(1-s)s] + d[3s^2]
\end{aligned} \tag{28}$$

At $s = 0$ and $s = 1$ we evaluate this to get the slopes at the endpoints:

$$f'(0) = 3(b-a) \qquad f'(1) = 3(d-c) \tag{29}$$

That is, up to a factor of 3, the derivatives of the function are related to the differences between coefficients.

Also take a look at where $B_{1,3}$ achieves its maximum on $[0, 1]$—with a bit of calculus, you should be able to show the maximum is at $s = 1/3$. Similarly, the maximum of $B_{2,3}$ is at $s = 2/3$. All the functions are non-negative over the interval. This leads us to think of the coefficients of $B_{1,3}$ and $B_{2,3}$ as approximating the values of the function at $s = 1/3$ and $s = 2/3$. The derivatives at the endpoints are in fact the slopes of the lines going through

$(0, a) \leftrightarrow (1/3, b)$ and $(2/3, c) \leftrightarrow (1, d)$, i.e. those lines are tangent to the curve!

This makes for a very intuitive control interface: we treat the interior coefficients simply as extra control points just like the ones at the endpoints of each interval. The big difference is that the spline won't necessarily interpolate these interior control points, but will follow them in a smooth way, with clear control over tangents or slopes at the endpoints of the interval. Since these interior control points are of the same quality as the endpoints (values of the function) we eliminate the problems we had with the Hermite basis.

Using this form of control points with the Bernstein basis leads to what is known as a *Bézier spline*, which is probably the most popular type of spline for animation.

3.7 The Bernstein Basis and its Wonderful Properties

The Bernstein basis generalizes to any degree polynomial, with the same kind of construction. We've already seen the linear Berstein basis,

$$\begin{aligned} B_{0,1}(s) &= 1 - s, \\ B_{1,1}(s) &= s, \end{aligned} \tag{30}$$

which is the natural choice for lerping. For degree k polynomials, we expand as before,

$$\begin{aligned} 1 &= [(1 - s) + s]^k \\ &= \sum_{j=0}^{k} \binom{k}{j} (1 - s)^{k-j} s^j, \end{aligned} \tag{31}$$

and pick these terms off as the Bernstein basis:

$$\begin{aligned} B_{j,k} &= \binom{k}{j} (1 - s)^{k-j} s^j \\ &= \frac{k!}{(k - j)! j!} (1 - s)^{k-j} s^j. \end{aligned} \tag{32}$$

You can confirm this fits the linear and cubic cases we've seen so far.

The quadratic Bernstein basis is also very useful:

$$\begin{aligned} B_{0,2} &= (1-s)^2, \\ B_{1,2} &= 2(1-s)s, \\ B_{2,2} &= s^2. \end{aligned} \tag{33}$$

Quadratic Bézier curves define the outlines of every glyph (letter, number, symbol) in TrueType fonts, for example. For this basis we think of the coefficient of $B_{1,2}$ as a control point at $s = 1/2$, the midpoint of the interval. The slope derivation works as with cubics: the quadratic curve interpolates the endpoints and is tangent to the lines connecting the endpoints to the middle control point.

The association of the coefficient of $B_{j,k}$ with an approximate function value at $s = j/k$ holds for all j and k. You can verify that $s = j/k$ is where the maximum of $B_{j,k}$ occurs with very little effort. With rather more effort, the Weierstrass approximation theorem can be proven using this: for any continuous function $f(s)$ on $[0, 1]$, the approximation

$$f(s) \approx \sum_{j=0}^{k} f(\tfrac{j}{k}) B_{j,k}(s) \tag{34}$$

converges uniformly to $f(s)$ as $k \to \infty$. This is a much stronger result than Taylor's theorem, which also deals with polynomial approximations to smooth functions, for example—the Bernstein approximation is guaranteed to be equally good over the entire interval $[0, 1]$ and only requires that f be continuous. This means, also, that even very high degree polynomials, when treated this way, can be used effectively to design curves—unlike interpolating polynomials.

The Bernstein approximation also happens to exactly interpolate constants (obviously—see our derivation from $1 = [(1-s)+s]^k$) and linear functions exactly. More precisely,

$$s = \sum_{j=0}^{k} (\tfrac{j}{k}) B_{j,k}(s) \qquad \forall s \in [0, 1] \tag{35}$$

This means uniform motion can be exactly matched without effort.

The Bernstein basis functions are always non-negative on $[0, 1]$, since $s \geq 0$ and $1 - s \geq 0$, and at every s sum to one (again, from our derivation). This means you can think of them as providing weights for a weighted average: the value of the spline at any point is a weighted average of the control point values, with the weights changing with s. In particular, the minimum and maximum of the spline can't overshoot the minimum and maximum of the control points, which is attractive for knowing that nothing can get too crazy: this is *not* the case for Catmull-Rom splines, for example, which can overshoot.

3.8 Evaluation

There is a neat trick available for evaluating a Bernstein polynomial called the *de Casteljau algorithm*, after its inventor, which provides better numerical stability—and is simply more elegant—than the brute-force approach. Essentially we can use a series of lerps to boil the input coefficients down to the final value, as explained below.

Let's start with a quadratic example, with coefficients f_0, f_1, and f_2, evaluated at $s \in [0, 1]$. Begin by lerping between f_0 and f_1 with parameter s, and between f_1 and f_2 with s as well:

$$\begin{aligned} f_0^1 &= f_0(1 - s) + f_1 s, \\ f_1^1 &= f_1(1 - s) + f_2 s. \end{aligned} \tag{36}$$

Then lerp between these intermediate values at parameter s as well:

$$f(s) = f_0^2 = f_0^1(1 - s) + f_1^1 s. \tag{37}$$

Expanding this reveals it to indeed be the Berstein polynomial at s:

$$\begin{aligned} f(s) &= [f_0(1-s) + f_1 s]\,(1-s) + [f_1(1-s) + f_2 s]\,s \\ &= f_0\left[(1-s)^2\right] + f_1\left[2(1-s)s\right] + f_2\left[s^2\right] \end{aligned} \tag{38}$$

This scheme generalizes to higher degrees too. For example, for cubics with coefficients f_0, f_1, f_2, and f_3, we first do the three lerps between adjacent coefficients:

$$\begin{aligned} f_0^1 &= f_0(1-s) + f_1 s, \\ f_1^1 &= f_1(1-s) + f_2 s, \\ f_2^1 &= f_2(1-s) + f_3 s. \end{aligned} \tag{39}$$

Then lerp between these:

$$\begin{aligned} f_0^2 &= f_0^1(1-s) + f_1^1 s, \\ f_1^2 &= f_1^1(1-s) + f_2^1 s. \end{aligned} \tag{40}$$

Finish off with one more lerp:

$$f(s) = f_0^3 = f_0^2(1-s) + f_1^2 s. \tag{41}$$

So simple!

But wait: it gets better. This can be interpreted geometrically too. The first set of lerps identify points on the line segments (at a certain ratio) between neighbouring control points. The second set of lerps identify points on the line segments between the first set of points, and so on. See the diagram in class for just how pretty this is.

3.9 Subdivision

Aside from adjusting control points, a critical part of artistic control over a motion curve is the ability to add new control points to an existing curve, gaining extra control. That is, we want to insert a new knot $t_{i+1/2}$ inside an existing interval $[t_i, t_{i+1}]$. The computer then needs to find new control point values: ideally the default new values should be such that the curve doesn't change at all (but simply exposes a new set of controls for finer-grained control).

In the piecewise-linear case, it's pretty obvious what the new control point should be: simply lerp between the old control points, taking

$$f_{i+1/2} = f_i \left(\frac{t_{i+1} - t_{i+1/2}}{t_{i+1} - t_i} \right) + f_{i+1} \left(\frac{t_{i+1/2} - t_i}{t_{i+1} - t_i} \right). \tag{42}$$

This divides the straight line segment on $[t_i, t_{i+1}]$ into two segments without changing the actual values of the function at all.

For higher degree polynomials, subdivision without changing the function is a bit trickier to arrange. Let's focus just on the unit interval again, relying on the transformation back to the real intervals to make it right.

Suppose our Bézier coefficients are $f_0, \ldots, f_k$ for a degree k polynomial on $[0, 1]$, and we want to split the interval at s. The amazing answer is that we reuse the intermediate coefficients generated in de Casteljau evaluation. The new coefficients on the $[0, s]$ subinterval should be $f_0, f_0^1, \ldots, f_0^k$; the new coeffients on the $[s, 1]$ subinterval should be $f_0^k, f_1^{k-1}, \ldots, f_{k-1}^1, f_k$.

Proving that this works is a little tedious, but not difficult at all for special cases such as $k = 2$ or $k = 3$ (which in practice is all we will care about for the course). I'll leave this as an exercise if you're curious.

3.10 Even More Splines

With Hermite or Bézier splines, it's easy for cubics to achieve C^1 smoothness: just make sure slopes match at the junctions between intervals. What about C^2? Notice that a piecewise-linear spline is C^0; if you integrate that you get a piecewise-quadratic which must be C^1 since its derivative is the piecewise-linear C^0 function. If you integrate again you get a piecewise-cubic which must be C^2. So we know it's possible to get C^2 smoothness with cubic splines—but short of doing a lot of integrals, how?

This isn't necessarily something we care about for motion curves: integrating a piecewise constant acceleration just gives a C^1 position function, for example, and that should be adequate for most dynamics. Most of the time Bézier cubics are just fine.

However, C^2 or higher smoothness can occasionally be useful for very smooth motion, like the path a camera should take without disturbing the audience. (More on camera paths in a later chapter.)

One possibility to achieve C^2 smoothness is to use Hermite or Bézier cubic splines as before, but instead of allowing the user to set slopes or interior control points on intervals, have the computer solve a large system of equations for values which match first and second derivatives at knots. This gives an interpolating C^2 spline. Unfortunately, whenever the user adjusts a control point, the system has to be solved again (which is potentially expensive), and more critically the entire curve can change. For just the regular C^1 form, when a user adjusts a control point it only affects the curve in the neighbouring intervals—the rest remains unchanged. This is called *local control*, and is a very good thing: the user can tweak one part of the motion without worrying that it's screwing up another part that was already good. The interpolating C^2 cubic instead has *global control*, where adjusting any control point has an effect on the entire curve from start to finish, which is bad.

To get C^2 piecewise-cubic splines with local control, it turns out you have to give up on the interpolating condition. Instead, you can build an *approximating* spline, which goes near but not necessarily through the control points. The foremost example of such a spline is called a *B-spline*.

B-splines actually generalize to any degree polynomial, and achieve higher and higher smoothness as the degree increases (at the expense of looser and looser approximation, and less and less local control). B-splines have been further generalized to *Non-Uniform Rational B-Splines* (NURBS for short) which is where the knots may be non-uniformly spaced, and instead of a cubic polynomial in each interval the curve uses a ratio fo two cubics—a rational function—both of which are themselves B-splines. B-splines and NURBS are extremely important for geometric modeling, especially when generalized from curves to surfaces, but we will not cover them in this course.

4 Exercises

4.1 Reading and Watching

You may find it useful to look up some of the mathematical constructs of this chapter in another resource, such as Wolfram Mathworld:

`http://mathworld.wolfram.com/`

4.2 Programming

Assignment 2 will involve writing a curve editor widget with PyQt, and using it to animate a simple scene. More details will come shortly.

4.3 Sample Exam Questions

This chapter is just brimming with math! In addition to knowing the terms in italics introduced in this chapter, expect both theoretical and calculational questions, such as:

- How would a C^1 piecewise-linear curve be constructed?
- Define the Bernstein basis polynomials for degree k.
- Consider a quadratic Bézier spline made of two intervals on knots $0, 1, 3$ with control point values $1, (5), 4, (3), 0$ where parentheses mark the controls interior to the two intervals. Is this spline C^0 or C^1?

- How do you subdivide a quadratic Bernstein polynomial on $[0, 1]$ at the midpoint $1/2$? Prove that the two new polynomials evaluate to be identical to the original.