A Blossoming Development of Splines

Copyright © 2006 by Morgan & Claypool

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews, without the prior permission of the publisher.

A Blossoming Development of Splines

Stephen Mann

www.morganclaypool.com

1598291165
9781598291162 paperback

1598291173
9781598291179 ebook

DOI 10.2200/S00041ED1V01200607CGR001

A Publication in the Morgan & Claypool Publishers Series
SYNTHESIS LECTURES ON COMPUTER GRAPHICS AND ANIMATION #1
Lecture #1
Series Editor: Brian A. Barsky, University of California, Berkeley

First Edition

10 9 8 7 6 5 4 3 2 1

A Blossoming Development of Splines

Stephen Mann
University of Waterloo
Canada

SYNTHESIS LECTURES IN COMPUTER GRAPHICS AND ANIMATION #1

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

In this lecture, we study Bézier and B-spline curves and surfaces, mathematical representations for free-form curves and surfaces that are common in CAD systems and are used to design aircraft and automobiles, as well as in modeling packages used by the computer animation industry. Bézier/B-splines represent polynomials and piecewise polynomials in a geometric manner using sets of control points that define the shape of the surface.

The primary analysis tool used in this lecture is blossoming, which gives an elegant labeling of the control points that allows us to analyze their properties geometrically. Blossoming is used to explore both Bézier and B-spline curves, and in particular to investigate continuity properties, change of basis algorithms, forward differencing, B-spline knot multiplicity, and knot insertion algorithms. We also look at triangle diagrams (which are closely related to blossoming), direct manipulation of B-spline curves, NURBS curves, and triangular and tensor product surfaces.

KEYWORDS

Bézier and B-splines curves and surface, Blossoming, Computer-aided geometric design, Splines, Triangular and tensor product spline surfaces

Contents

Preface

These are a subset of course notes that I started developing in 1993 for the University of Waterloo course on splines. I wanted a blossom development of the spline material, and found no reference adequate for that purpose. Although some possible choices have appeared since then [10, 9], I preferred the material that I had developed. The notes you see here are mostly restricted to the blossoming material that appears in my course notes, although there are a few side trips to emphasize some important points.

A word about the style and intended audience: this lecture is not aimed at mathematicians. Instead, it is aimed at senior undergraduates or first-year graduate students in computer science, whose mathematics is a bit weak or perhaps a bit rusty. Exposure to calculus and linear algebra is expected, but I try to include a brief reminder of the mathematical ideas needed in the text. The proofs are informal and could be tightened up a lot, but I have tried to write them in a style that is more useful to someone that needs a bit more guidance. Further, there is a varying amount of rigor: sometimes, proofs are omitted or glossed over, at other times, the proofs are done in more detail. And the exposition tends to be a bit chatty. Again, the intent is to provide a level of detail that will allow the intended audience to absorb and appreciate the material.

Two sources were the main inspiration for these notes: Lyle Ramshaw's tech report, and the papers and talks that were the basis for Ron Goldman's book, *Pyramid Algorithms*. Both are excellent supplements to these notes. Because of the way these notes were developed, the references are a bit skimpy; my apologies to anyone whose work I should have cited—I would be interested in hearing from you to add an appropriate citation to future versions of these notes.

I am indebted to Lyle Ramshaw and an anonymous reviewer, whose comments on a draft of these notes allowed me to expand several portions from short notes to myself (to elaborate on in class) into a readable text.

CHAPTER 1

Introduction and Background

Polynomial curves and surfaces have a long mathematical history. In CAD and modeling packages, Bézier and B-spline curves and surfaces have been the standard modeling techniques for several decades. Although there are now other competing methods (subdivision surfaces, triangle meshes), the Bézier/B-spline form is still in common use.

In this set of notes, I give a geometrical introduction to Bézier and B-spline curves and surfaces using *blossming*, a technique that allows us to label the Bézier and B-spline control points (and intermediate points of computation) so that we can visualize the mathematics.

I begin with some background mathematics on affine geometry. Then the next chapter presents polynomials curves, using *blossoming* as the main analysis tool. The following chapter extends these ideas to piecewise polynomial curves and B-splines. Finally, the last chapter looks at triangular and tensor product surfaces.

1.1 MATHEMATICAL BACKGROUND

I give a brief sketch of some of the mathematical ideas that we will use in this lecture; see linear algebra and geometry text books for additional details.

One of the core ideas used in splines is that of a vector space, the properties of which are as follows.

Definition 1.1. *Vector space: A non-empty set V and two operators:*

- $V \times V \to V : (\vec{u}, \vec{v}) \mapsto \vec{u} + \vec{v}$
- $\mathcal{R} \times V \to V : (\alpha, \vec{v}) \mapsto \alpha\vec{v}$

such that for $\vec{u}, \vec{v}, \vec{w} \in V$ and $a, b \in \mathcal{R}$ (where $\mathcal{R}$ are the real numbers) the following hold:

1. *commutativity of $+$*
2. *associativity of $+$*
3. *unique additive identity $\vec{0}$ exists*
4. *additive inverse of $\vec{v}$ exists*

5. *1 is a multiplicative identity*
6. *left and right distributivity of multiplication*
7. $a(b\vec{v}) = (ab)\vec{v}$

We will also use the ideas of spans and bases, where a *span* is a set of vectors S in a vector space V where any element of V can be written as a linear combination of the elements of S, and a *basis* is a smallest set of such vectors. Note that a basis for an n-dimensional vector space has exactly n elements, and that a set of n vectors in an n-dimensional vectors space form a basis for the space if and only if $\vec{0}$ has a unique representation relative to the set of basis vectors (see Exercise 1 of Section 1.1.2).

Although vector spaces are important in splines (primarily because we need the ideas of polynomial vector spaces and polynomial bases), they are insufficient for describing the geometry we wish to work with. Instead, we will work in affine spaces.

Definition 1.2. *An affine space $\mathcal{A} = \{P, V\}$ is a set of points P and vectors V where V forms a vector space and where the following operations for points exist:*

- $V \times P \to P : (\vec{v}, p) \mapsto \vec{v} + p$
- $P \times P \to V : (p, q) \mapsto p - q$ *where* $q + (p - q) = p$

By abuse of notation, we get *affine combinations*: $\sum_{i=1}^{n} a_i p_i = p \in P$ is said to be an affine combination if $\sum a_i = 1$. We will also allow $\sum a_i = 0$ (to be interpreted as a vector). If $\sum a_i \notin \{0, 1\}$ then the expression is invalid (i.e., it has no geometric meaning).

A *linear transformation* preserves linear combinations while an *affine transformation* preserves affine combinations, i.e.,

$$L\Big(\sum a_i \vec{v}_i\Big) = \sum a_i L(\vec{v}_i)$$

is a linear transformation for all $a_i \in \mathcal{R}$, and

$$A\Big(\sum a_i p_i\Big) = \sum a_i A(p_i), \quad \sum a_i = 1$$

is an affine transformation.

Note that every linear transformation on V is an affine transformation on P. Intuitively, affine transformations are the ones we know about: scale, rotate, translate, shear, etc. Translation (and its composition with the other linear transformations) is the only additional affine transformation on P above and beyond those that are linear transformations on V.

As an example of an affine combination, we note that a line can be described as $l(t) = (1-t)p + tq$, where $p, q \in P$ and we let t vary from $-\infty$ to ∞. This leads to the following two observations:

1. Affine transformations map lines to lines.
2. Affine transformations preserve ratios of distances along a line (more generally, along parallel lines).

Claim 1.1. *If a point transformation preserves ratios of parallel distances, then it is affine.*

Proof. *Exercise 4 of Section 1.1.2*

Note that we have defined our line $l(t)$ as a *parametric line*. The points p and q typically have x, y, and possibly z coordinates (in fact, p and q can be of any dimension), and the domain parameter t is in a different space. The parametric representation looks a bit different from the more familiar form of a line $y = mx + b$. However, in geometric modeling, the parametric representation is more useful, since we can represent all lines (including vertical lines) with a single representation, and the non-coordinate parametric line definition frees us from being tied to a particular coordinate system.

1.1.1 Why Affine Geometry?

While vector spaces are useful for many things, for geometric modeling, they are inadequate. In geometric modeling, we want to describe our objects "simply." In particular, we want to describe our objects as a function of a small number of "representatives" (usually points, but some times a mix of points and vectors), and then transform the representatives to transform the entire object. For example, in a modeling package, you might draw a curve on the screen, and then move it to a different location on the screen and then rotate it. By having a "nice" representation of the curve, we can achieve this translation/rotation by just translating/rotating the curve representatives.

To see why a vector space representation is inadequate, suppose we are working in a vector space, and we describe our object with vectors $\vec{u}$ and $\vec{v}$. Further suppose that the vector $\vec{u} + \vec{v}$ is on our object (Fig. 1.1, left). Now we decide to translate our object by a vector $\vec{t}$. In a vector space, it is not obvious how to translate something, since vectors are "free"; however, something that seems similar to translation by $\vec{t}$ is to add $\vec{t}$ to the representatives, thus mapping $\vec{v}$ to $\vec{v} + \vec{t}$. Our transformed representation for our object is $\vec{U} = \vec{u} + \vec{t}$ and $\vec{V} = \vec{v} + \vec{t}$. But now (letting T represent the translation by $\vec{t}$), we find that we cannot transform our object by just transforming

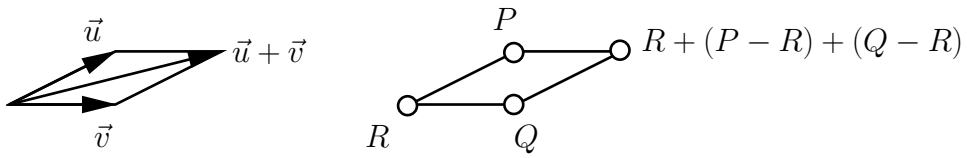


FIGURE 1.1: Vector space construction versus affine construction

the describing elements $\vec{u}$, $\vec{v}$, since

$$\begin{aligned} T(\vec{u}+\vec{v}) &= \vec{u}+\vec{v}+\vec{t} \\ &\neq T(\vec{u})+T(\vec{v}) \\ &= \vec{u}+\vec{v}+2\vec{t} \end{aligned}$$

In the affine setting, if our representative objects are points P and Q, we translate them by adding the vector $\vec{t}$ to them, mapping P to $P+\vec{t}$ and Q to $Q+\vec{t}$. However, in an affine setting, the entity of $P+Q$ does not have geometric meaning. If we had a third point, R, and we wanted the point $R+(P-R)+(Q-R)$ as a point on our object (Fig. 1.1, right), we now see that

$$\begin{aligned} T(R+(P-R)+(Q-R)) &= R+(P-R)+(Q-R)+\vec{t} \\ &= T(R)+T(P-R)+T(Q-R) \end{aligned}$$

since vectors are unchanged under translation. Thus, in the affine setting, translation has a well-defined meaning that is preserved by affine combinations, something that is lacking in the linear setting.

Clearly, there was more structure in the affine setting than in the linear setting in the example in the previous paragraph. While constraints could be layered on top of a vector space to restrict vector combinations to ones that behave well under translation, those constraints basically make the vector space into an affine space, so conceptually we chose instead to work directly in an affine space, rather than in a linear space with some extra rules attached.

1.1.2 Exercises

1. Prove that n vectors in an n-dimensional vectors space form a basis for that space if and only if $\vec{0}$ has a unique representation relative to this set of vectors.
2. Show that parallel lines map to parallel lines under an affine transformation.
3. Show that perpendicularity is not preserved by affine transformations.
4. Prove that if a point transformation preserves ratios of parallel distances, then it is affine.

CHAPTER 2

Polynomial Curves

You should be familiar with the monomial form of a scalar-valued, degree n polynomial,

$$F(u) = \sum_{i=0}^{n} a_i u^i$$

where $a_i \in \mathcal{R}$.

If we consider the set P_n of degree n polynomials, then P_n forms a vector space of dimension $n+1$. You can prove this by showing that all the properties of addition and scalar multiplication required by a vector space hold (Exercise 1 of Section 2.2.1).

Proposition 2.1. *$\{u^i \mid i = 0, \ldots, n\}$ forms a basis for P_n. (That is, if F is a polynomial, then we can write F as $\sum_{i=0}^{n} a_i u^i$ for $a_i \in \mathcal{R}$. This is the monomial representation of F.)*

Proof. *We have the result if we can show that $\vec{0} = \sum_{i=0}^{n} a_i u^i$ if and only if $a_i = 0$ for all i. Clearly, $a_0 = 0$ (otherwise our polynomial is nonzero at $u = 0$). We can prove that $a_i = 0$ by differentiation.*

Note that the coefficients of a polynomial do not have to be scalar valued. We could instead use an array of scalars (i.e., $a_i \in \mathcal{R}^d$). The polynomial function would then give back an array of scalars. In geometric modeling, we would like the coefficients of our polynomials to have geometric meaning. One way to achieve this with monomials is to make a_0 (the coefficient of 1) be a point P and the remaining a_i be vectors, $\vec{v}_i$:

$$F(u) = P + \sum_{i=1}^{n} \vec{v}_i u^i$$

Further, the vector coefficients are related to the derivatives of F at $u = 0$; take the derivatives of F to see the exact relationship (see also the proof of Theorem 2.2 and Section 2.7).

There are other bases for the space of degree n polynomials. While the monomials have the nice properties that they are simple in form and fast to evaluate, we will study other polynomial bases that have properties that are more desirable for modeling purposes. The following is an example of another polynomial basis.

Proposition 2.2. *For $t_0, \ldots, t_n \in \mathcal{R}$ pairwise distinct,*

$$L_i^n(u) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{u - t_j}{t_i - t_j}, \quad i = 0, \ldots, n$$

is a basis for P_n.

Proof. *Assume $F(u) \equiv 0$ where $F(u) = \sum_{i=0}^{n} c_i L_i^n(u)$. For t_i, $L_i^n(t_j) = \delta_{ij}$, where δ_{ij} is 1 if $i = j$ and 0 otherwise. Thus,*

$$\sum_{i=0}^{n} c_i L_i^n(t_j) = c_j \quad \Rightarrow c_i = 0$$

These L_i^n are the *Lagrange polynomials*.

It turns out that the degree n Lagrange polynomials sum to 1: $\sum L_i^n(u) = 1$ (Exercise 2 of Section 2.2.1). This means that if we use the degree n Lagrange polynomials as weights to $n + 1$ points, then we have an affine combination of points, so the expression

$$\sum_{i=0}^{n} P_i L_i^n(t_j)$$

has geometric meaning in an affine space. In their simplest form, if we weight two points with the degree 1 Lagrange polynomials, then we have a parametric line:

$$l(u) = P_0 \frac{u - t_1}{t_0 - t_1} + P_1 \frac{u - t_0}{t_1 - t_0}$$

For higher degrees, if we use points as the coefficients to the Lagrange polynomials, then we get a *parametric curve* that interpolates those points. By parametric, we mean that the curve has a domain that is independent of the range. This allows us to create curves that are not graphs of polynomial functions; instead, the curve can wrap around itself, etc.

Thus, Lagrange polynomials give us a method for creating curves that interpolate a set of points. While this may seem ideal for geometric modeling, it turns out that there are problems with interpolating curves. At high degrees, interpolatory polynomial curves exhibit poor behavior, and are generally not used; see Fig. 2.1 for some examples. In these examples, the t_i values (also called *knots*) are 0, 1, 2, ... For interpolatory problems of this nature, we usually use piecewise polynomial interpolatory curves instead of a single polynomial to interpolate this data. (Although we will study piecewise polynomial curves in later sections, we will not study interpolatory piecewise polynomial curves; see [7] for a discussion of piecewise polynomials interpolation techniques.)

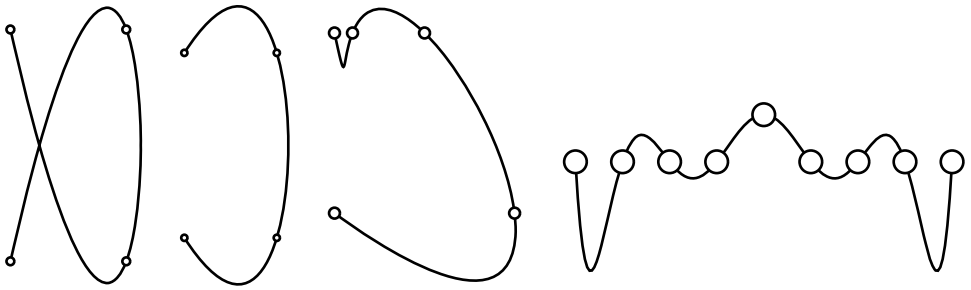


FIGURE 2.1: Interpolatory curves using uniform knots

2.1 IMPLEMENTATIONS

1. Implement an interactive 2D Lagrange curve editor with the following functionality:
   - The left mouse button adds a new interpolation point.
   - The middle mouse button is used to move points.
   - There are two display modes (accessible through a menu):
     - Just the curve.
     - The curve and the interpolation points.
   - There should be a reset key/menu option that clears all the control points.

If $n+1$ interpolation points have been entered, then draw a degree n Lagrange interpolatory curve using a uniform knot vector with the curve passing through the points in the order in which they were created.

2.2 BERNSTEIN POLYNOMIALS AND BÉZIER CURVES

A third polynomial basis that we will consider is the Bernstein basis, which is used to form Bézier curves.

Definition 2.1. *(Bernstein polynomials):* $B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$.

Proposition 2.3. *The degree n Bernstein polynomials form a basis for* P_n.

Proof. Exercise 3 of Section 2.2.1

In the Bernstein basis, we usually work with the polynomials over the interval $[0, 1]$. Sometimes, however, we will want to work with the Bernstein basis over another interval. This leads to the generalized Bernstein basis.

Definition 2.2. *(Generalized Bernstein polynomials over $[s, t]$):*

$$B_i^n(u) = \frac{1}{(t-s)^n}\binom{n}{i}(u-s)^i(t-u)^{n-i}$$
$$= \binom{n}{i}\left(\frac{u-s}{t-s}\right)^i\left(\frac{t-u}{t-s}\right)^{n-i}$$

Proposition 2.4. *Over the interval* $[0, 1]$,

1. $\sum_{i=0}^n B_i^n(u) = (u + (1-u))^n$,
2. $\sum_{i=0}^n B_i^n(u) = 1$,
3. $B_i^n(u) \geq 0, \quad 0 \leq u \leq 1$,
4. $B_i^n(u) = u B_{i-1}^{n-1}(u) + (1-u) B_i^{n-1}(u)$ *(recurrence relation)*,
5. $B_i^n(1-u) = B_{n-i}^n(u)$ *(symmetry)*,
6. $B_i^n(u)$ *has local maximum at* i/n.

Proof.

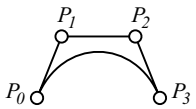
1. $(a+b)^n = \sum_{i=0}^n \binom{n}{i} a^i b^{n-i}$ *(binomial theorem);*
2. *Follows from above.*
3. $u \geq 0$ *over* $[0, 1]$ *and* $1 - u \geq 0$ *over* $[0, 1]$ *and binomial coefficient is always positive, so* $B_i^n(u) \geq 0$.
4. $$B_i^n(u) = \frac{n!}{i!(n-i)!}\left(\frac{n-i}{n} + \frac{i}{n}\right)u^i(1-u)^{n-i}$$
$$= \frac{(n-1)!}{i!(n-i-1)!}u^i(1-u)^{n-i} + \frac{(n-1)!}{(i-1)!(n-i)!}u^i(1-u)^{n-i}$$
$$= \binom{n-1}{i}u^i(1-u)^{n-i-1}(1-u) + \binom{n-1}{i-1}u^{i-1}(1-u)^{n-i}u$$
$$= (1-u)B_i^{n-1}(u) + u B_{i-1}^{n-1}(u)$$
5. $$B_i^n(1-u) = \binom{n}{i}(1-u)^i(1-(1-u))^{n-i}$$
$$= \binom{n}{n-i}(1-u)^{n-(n-i)}u^{n-i}$$
$$= B_{n-i}^n(u)$$
6. *Differentiate (Exercise 4 of Section 2.2.1).*

Similar propositions hold for the generalized Bernstein polynomials over the interval $[s, t]$.

Having defined the Bernstein basis, we can now look at Bézier curves. Over the interval [0, 1], a degree n Bézier curve is given by

$$B(t) = \sum_{i=0}^{n} P_i B_i^n(t)$$

where P_i are *control points*. As we vary t from 0 to 1, we trace out a curve:



The piecewise linear curve connecting the control points in sequence is known as the *control polygon*.

Immediately from the defining equation of Bézier curves, we see that they have an extremely useful property that is lacking in the monomials: the basis functions (the Bernstein polynomials) sum to 1, meaning that the curve is an affine combination of its coefficients. This allows us to treat the coefficients as points in a strong affine sense, and the curve has the desired affine geometric properties. In addition, Bézier curves have many other nice properties that make them useful for modeling. The remainder of this chapter investigates these other properties of Bézier curves.

It is easy to see from the definition of the Bernstein polynomials that

1. $B(0) = P_0$
2. $B(1) = P_n$

A Bézier curve does not, in general, interpolate any of its other control points. However, these other control points can be used to compute the derivatives of a Bézier curve as we will see later.

Although we can evaluate a Bézier curve by evaluating the Bernstein polynomials and then using them to weight the control points, the de Casteljau algorithm is another method for evaluating these curves.

de Casteljau's algorithm uses repeated linear interpolation to evaluate a Bézier curve. The idea is to compute $P_j^1 = (1-t)P_j + tP_{j+1}$ for $j = 0, \ldots, n-1$. We then repeat this process to compute P_j^2 from P_j^1. Continuing, we compute a sequence of sets of points P^1, P^2, etc., each set having one fewer points than the previous set, until we reach P^n, which has a single point. This point turns out to be a point on the curve. This process is illustrated for a cubic curve in Fig. 2.2, with $t = 0.5$.

We can also consider the dataflow of the de Castlejau algorithm. The dataflow for the cubic algorithm is illustrated in Fig. 2.3 (left); such a diagram is also known as a *triangle diagram*.

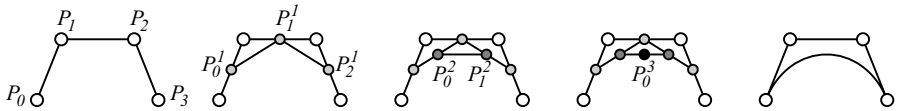


FIGURE 2.2: The de Casteljau algorithm

We start with the control points at the bottom of the figure. Each node in the diagram indicates one of the values computed by de Casteljau's algorithm. The arcs into an intermediate node show which points are combined to create the intermediate node, and the labels of the arcs give the weights used to combine these points.

A note about the triangle diagram. Suppose we leave the labels on the edges unchanged, put a 1 at the top, and run the triangle diagram backwards (Fig. 2.3, right). When we sum all paths from the top to a leaf node, we get a Bernstein polynomial at the leaf node, and we find that the $n + 1$ leaf nodes will have a complete set of the degree n Bernstein polynomials. In general, running the triangle diagram backwards gives us the weight functions for the control points. We will revisit triangle diagrams after we have blossoming, and see that they are a powerful analysis technique.

The code for de Casteljau's algorithm is quite simple. First, we observe from the triangle diagram that after we combine P_0 and P_1 to form P_0^1, we no longer need the point P_0. Thus, we can store the result of $(1 - t)P_0 + tP_1$ back in P_0. In a similar manner, we can reuse the storage of all the control points. This leads to the following pseudo-code for de Castlejau's algorithm:

```
for i = n to 1
  for j = 0 to i-1
    P_j = (1-t)P_j + t P_{j+1}
  end
end
return P_0
```

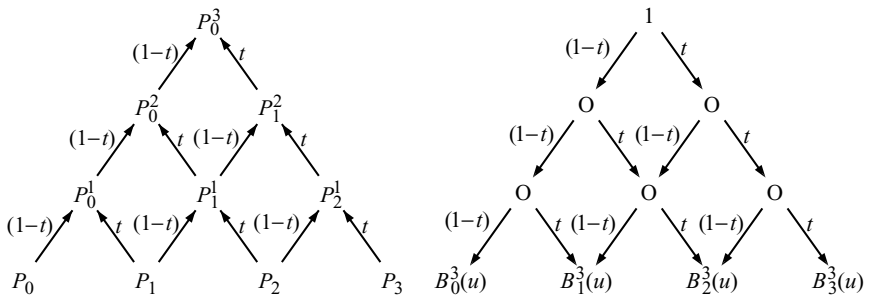


FIGURE 2.3: Triangle diagrams of de Casteljau's algorithm

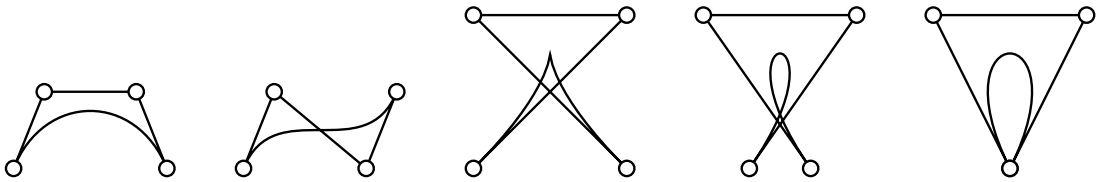


FIGURE 2.4: Examples of cubic Bézier curves; in the last example, the bottom control point is both the first and the last control point

Thus, while the algorithm has $O(n^2)$ running time, it only requires $O(n)$ storage. Also note that to compute a sequence of points on the curve, we will need to copy the control points before running this version of de Casteljau's algorithm.

Some examples of cubic Bézier curves and their control polygons are shown in Fig. 2.4. In these examples, we see that parametric cubic curves can have inflection points, cusps, and self-intersections.

It is straightforward to show that Bézier curves have a variety of useful properties; most of these results come directly from the properties of the Bernstein polynomials.

- Affine invariance: $T(\sum_i P_i B_i^n(u)) = \sum_i T(P_i) B_i^n(u)$.
  True because the basis functions sum to 1.
- Invariance under affine parameter transformations: Clear from de Casteljau's algorithm.
- Convex hull property: Curve lies within the convex hull of its control points as it is a convex combination of its control points.
  (The *convex hull* of a set of points is the smallest polygon that contains all of the points; a *convex combination* of a set of points is an affine combination where all the weights are nonnegative.)
- Endpoint interpolation: Clear.
- Symmetry: A result of the symmetry in the Bernstein polynomials.
- Linear precision: The following is an identity:

$$\sum_{i=0}^{n} \frac{i}{n} B_i^n(t) = t$$

  This identity tells us that if we place our control points uniformly along a line, then the curve will be a line (with linear parameterization). We will revisit this later once we look at degree raising; see Section 2.3.2.
- Variation diminishing property (We will not prove this one; see [10]): Intersect a planar Bézier curve with a line. Count the number of intersections between the line and the

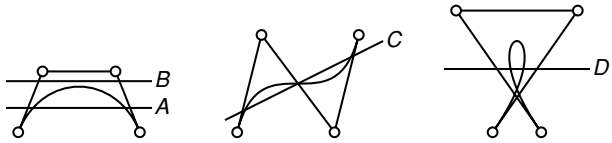


FIGURE 2.5: Lines intersecting Bézier curves and their control polygons

curve and between the line and the control polygon. The variation diminishing property says that the number of intersections with the curve is no more than the number of intersections with the control polygon.

A bit more obscure than the other properties, the variation diminishing property allows us to bound the variation in the curve by looking at the variation in the control polygon. For example, if the control polygon "wiggles" w times, then the curve will "wiggle" no more than w times.

Figure 2.5 shows some examples illustrating this property. We see that lines A, C, and D all intersect the Bézier curve and the control polygon in equal numbers of points. Note that it is possible for a line to intersect the control polygon in more points that it intersects the curve (line B is an example in this figure); the variation diminishing property only says that the opposite will not occur.

Before studying further properties of Bézier curves, we will look at the idea of blossoming.

2.2.1 Exercises

1. Prove that P_n^d is a vector space.
2. Show that the degree n Lagrange polynomials sum to 1.
3. Show that the degree n Bernstein polynomials form a basis for the space of degree n polynomials.
4. Show that the Bernstein polynomial $B_i^n(t)$ reaches its maximum at $t = i/n$. Find the maximum value.
5. Express the polynomial $f(x) = x^3 + x^2 + x + 1$ in the Bernstein basis.
6. The change of basis from the monomial basis to the Bernstein basis is a linear transformation. Derive this matrix for cubic polynomials.

2.3 BLOSSOMING

In this section, I state and prove the *blossoming theorem*, which will be the basis of much of the analyses in the remainder of this lecture. The theorem and its application are more important than is the proof of the theorem, so you may want to skip the proof on a first reading.

We start by defining a few terms used in the blossoming theorem.

Definition 2.3. *Let* $f : (\mathcal{R}^c)^n \mapsto \mathcal{R}^d$

1. f *is multiaffine if* $f(a_1, \ldots, a_{i-1}, v, a_{i+1}, \ldots, a_n)$ *is affine in* v.
2. f *is symmetric if* $f(u_1, \ldots, u_n) = f(u_{\sigma(1)}, \ldots u_{\sigma(n)})$ *for all permutations* σ.

Intuitively, the multiaffine property is equivalent to each variable appearing to no more than the power 1, while the symmetry property allows us to change the order of the arguments without changing the value of the function.

Example 2.1. *If* $f(u, v, w)$ *is multiaffine, then* $f(u, 0, 0)$ *is affine in* u.

Example 2.2. *If* $f(u, v, w)$ *is symmetric, then* $f(a, b, c) = f(a, c, b) = f(b, a, c) = f(b, c, a) = f(c, a, b) = f(c, b, a)$.

Theorem 2.1. *The blossoming principle*
Let $F : \mathcal{P} \mapsto \mathcal{Q}$ *be a degree* n *polynomial, where* $\mathcal{P}$ *and* $\mathcal{Q}$ *are affine spaces.*
Then there exists a unique map $f : \mathcal{P}^n \mapsto \mathcal{Q}$ *such that*

1. f *is symmetric,*
2. f *is multiaffine,*
3. $f(u, \ldots, u) = F(u)$.

f *is said to be the* polar form *or multiaffine blossom of* F.

Proof. *For simplicity, let* $\mathcal{P} = \mathcal{R}$ *(you will generalize this proof for* $\mathcal{P}$ *of arbitrary dimension in Exercise 1 of Section 4.1.2). Then, each term* T_F *of* $F(u)$ *is of the form* $c_i u^i$, *where* $i \leq n$, *and where* $c_i \in \mathcal{R}$.

Suppose we wish to construct $f(u_1, \ldots, u_n)$ *from* $F(u) = \sum_{i=0}^{n} c_i u^i$. *It is easy to convert* F *into an* n*-variate function* $f^1(u_1, \ldots, u_n)$*: for each term* $c_i u^i$ *of* F, *place a corresponding term* $c_i u_1 \cdot \cdots \cdot u_i$ *in* f^1. *Clearly,* $f^1(u, \ldots, u) = F(u)$.

But f^1 *is not symmetric. Thus, we symmeterize each term of* f^1 *into a term of* f, *converting the term* $c_i u_1 \cdot \cdots \cdot u_i$ *into*

$$\frac{c_i}{\binom{n}{i}} \sum_{\sigma} u_{\sigma(1)} \cdot \cdots \cdot u_{\sigma(i)}$$

where σ *is a permutation of the integers* $1, \ldots, n$ *with* $\sigma(1) < \cdots < \sigma(i)$ *and with* $\sigma(i+1) < \cdots < \sigma(n)$, *leading to the* $\binom{n}{i}$ *coefficient. This symmeterization retains the diagonal property, and the resulting* f *is multiaffine as each variable is of no more than degree 1 in each term of* f.

Uniqueness is harder. While it is clear that each f *corresponds to a unique* F *(due to agreeing on the diagonal), it might be possible for some* F *to map to two different blossoms. We first prove that*

f is completely defined by

$$f(\underbrace{0,\ldots,0}_{n-i},\underbrace{1,\ldots,1}_{i}) \quad \forall i = 0,\ldots,n$$

Here, by "complete," I mean that given the above blossom values, we can find $f(u_1,\ldots,u_n)$ for any $u_1,\ldots,u_n$. We sketch a proof by induction. If all of the blossom arguments are 0 and 1, then there is nothing to prove. The inductive hypothesis is that if $n-j$ out of n of the blossom arguments are 0 and 1, then we can express this function in terms of blossom evaluations where all the arguments are 0 and 1. Now suppose only $n-j-1$ of the arguments are 0 and 1. Then

$$\begin{aligned} f(u_1,\ldots,u_{j+1},\underbrace{0,\ldots,0}_{k},\underbrace{1,\ldots,1}_{n-j-k-1}) &= f(u_1,\ldots,u_j,(1-u_{j+1})\cdot 0+u_{j+1}\cdot 1,\underbrace{0,\ldots,0}_{k},\underbrace{1,\ldots,1}_{n-j-k-1}) \\ &= (1-u_{j+1})f(u_1,\ldots,u_j,\underbrace{0,\ldots,0}_{k+1},\underbrace{1,\ldots,1}_{n-j-k-1}) \\ &\quad +u_{j+1}f(u_1,\ldots,u_j,\underbrace{0,\ldots,0}_{k},\underbrace{1,\ldots,1}_{n-j-k}), \end{aligned}$$

and by our inductive hypothesis, we can express each of these blossom evaluations with arguments sets having only 0s and 1s.

To prove uniqueness, suppose $F(u) = f(u,\ldots,u) = g(u,\ldots,u)$. Consider $f(u,\ldots,u)$:

$$\begin{aligned} f(u,\ldots,u) &= f((1-u)\cdot 0+u\cdot 1,u,\ldots,u) \\ &= (1-u)f(0,u,\ldots,u)+uf(1,u,\ldots,u) \\ &\vdots \\ &= \sum_{i=0}^{n}\underbrace{\binom{n}{i}(1-u)^{n-i}u^i}_{Bernstein} f(\underbrace{0,\ldots,0}_{n-i},\underbrace{1,\ldots,1}_{i}) \\ g(u,\ldots,u) &= \sum_{i=0}^{n}\binom{n}{i}(1-u)^{n-i}u^i g(\underbrace{0,\ldots,0}_{n-i},\underbrace{1,\ldots,1}_{i}) \\ \Rightarrow f(0,\ldots,0,1,\ldots,1) &= g(0,\ldots,0,1,\ldots,1) \qquad \textit{as Bernsteins form a basis} \\ \Rightarrow f &= g \end{aligned}$$

Note that this also tells us that the Bézier control points have the blossom values $f(0,\ldots,0,1,\ldots,1)$.

Example 2.3. *Let $F(u) = 3u^3+2u^2+6u+1$. We can blossom each term and then sum these blossoms to construct $f(u_1,u_2,u_3)$:*

- $1 \mapsto 1$
- $6u \mapsto 6(u_1+u_2+u_3)/3$
- $2u^2 \mapsto 2(u_1u_2+u_2u_3+u_3u_1)/3$
- $3u^3 \mapsto 3u_1u_2u_3$

Thus,

$$f(u_1, u_2, u_3) = 3u_1u_2u_3 + 2(u_1u_2 + u_2u_3 + u_3u_1)/3 + 2(u_1 + u_2 + u_3) + 1.$$

By inspection, we see that $f(u, u, u) = F(u)$.

How do f *and* F *relate to the Bézier control points? First pick an interval of interest (say* $[0, 1]$ *for simplicity). Next, evaluate* f *at* $f(0, 0, 0) = 1$, $f(0, 0, 1) = 3$, $f(0, 1, 1) = 5\frac{2}{3}$ *and* $f(1, 1, 1) = 12$. *These are the Bézier control points of* F. *Thus*,

$$F(u) = 1 \cdot (1 - u)^3 + 3 \cdot 3u(1 - u)^2 + 5\frac{2}{3} \cdot 3u^2(1 - u) + 12u^3$$

If we wanted a curve in two space, then we would need two parametric functions, $F_x(u)$ and $F_y(u)$, we would blossom both, giving f_x and f_y, evaluate these blossoms at the appropriate 0–1 argument mixes to get the control points, and then weight these paired values with the Bernstein polynomials to get a Bézier curve.

It may seem strange that we have taken a function of one variable and replaced it with a function of many variables, which would seem to be more complicated. But what we have really done is taken a high-degree function of one variable and exchange it for a function of many variables *that is linear in each variable*. It is the linearity in each variable that makes the blossom valuable to us, as illustrated by revisiting de Casteljau's algorithm.

2.3.1 de Casteljau Revisited

Suppose we have a cubic curve F defined over $[0, 1]$. Consider the blossom f of F, and the points $f(0, 0, 0)$, $f(0, 0, 1)$, $f(0, 1, 1)$, $f(1, 1, 1)$.

Consider $(1 - t)f(0, 0, 0) + tf(0, 0, 1)$. Since f is multiaffine, this is equal to $f(0, 0, t)$, as $t = 0 \cdot (1 - t) + 1 \cdot t$:

$$(1 - t)f(0, 0, 0) + tf(0, 0, 1) = f(0, 0, t)$$

Repeating this process and labeling the control points and the intermediate points with their blossom values, we obtain Fig. 2.6. Thus, we see again that the Bézier control points are given by special values of the blossom of the Bézier curve.

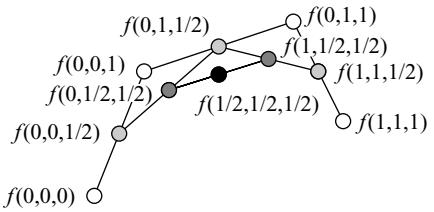


FIGURE 2.6: de Castlejau's algorithm with blossom labels

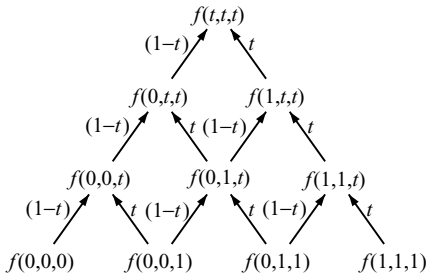


FIGURE 2.7: Triangle diagram of de Casteljau algorithm with blossom labels

In Fig. 2.6, consider the points $f(0, 0, 0)$, $f(0, 0, t)$, $f(0, t, t)$, and $f(t, t, t)$. If we treat these as Bézier control points, we see that we obtain the same curve as F, only defined over the interval $[0, t]$ rather than on $[0, 1]$. Likewise, the points $f(t, t, t)$, $f(t, t, 1)$, $f(t, 1, 1)$, and $f(1, 1, 1)$ are the control points of the portion of F over the interval $[t, 1]$. Thus, the de Casteljau algorithm not only evaluates a Bézier curve, it also *subdivides* the curve into two pieces. If we look at the triangle diagram for Bézier curves (Fig. 2.7) we see that these subdivision points appear along the "outside edges" of the triangle diagram.

Subdivision is a useful operation for a variety of tasks, beyond the obvious one of when you want a subportion of the curve. As we will see when we look at derivatives, subdivision gives us the derivatives of the curve at the point of evaluation. Subdivision also has the useful property that the control polygons of the subdivided curve segments lie closer to the curve than do the unsubdivided segments. In the limit, as we subdivide an infinite number of times, the subdivided control polygons converge to the curve itself. This convergence property can be used to make an adaptive drawing algorithm for Bézier curves, where you stop subdividing when the control points are nearly colinear or when all of them lie within a pixel. Typically, though, such a subdivision drawing algorithm is a bit of an overkill—sampling the curve a fixed number of times (for a simple curve segment, 20 samples should suffice) and drawing the polyline connecting those points is faster than is the adaptive algorithm and sufficient for rendering purposes.

2.3.2 Degree Raising

Suppose we have a degree 2 Bézier curve $F(u)$, and we wish to know its degree 3 representation. For monomials, this question is trivial. But for the Bézier representation, we need to find a new set of control points.

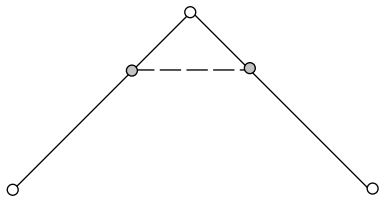
We know $F(u)$ and its blossom $f(u, v)$. What we want is $g(u, v, w)$, the trivariate blossom of F. The trivariate blossom $g(u, v, w)$ is a symmetric trivariate function that agrees with F on the diagonal. The following clearly works:

$$g(u, v, w) = \frac{f(u, v) + f(v, w) + f(w, u)}{3}$$

The control points of G are given by $g(0,0,0)$, $g(0,0,1)$, $g(0,1,1)$, $g(1,1,1)$:

- $g(0,0,0) = (f(0,0) + f(0,0) + f(0,0))/3 = f(0,0)$
- $g(0,0,1) = (f(0,0) + f(0,1) + f(1,0))/3 = f(0,0)/3 + 2f(0,1)/3$
- $g(0,1,1) = (f(0,1) + f(1,1) + f(1,0))/3 = 2f(0,1)/3 + f(1,1)/3$
- $g(1,1,1) = (f(1,1) + f(1,1) + f(1,1))/3 = f(1,1)$

Looking at the control polygon, we see the following relationship between the degree 2 and degree 3 control points:



This formula can be generalized to arbitrary degree:

$$P_i^{n+1} = \frac{i}{n+1} p_{i-1}^n + \frac{n+1-i}{n+1} P_i^n, \quad i = 0, \ldots, n+1$$

Geometrically, we break each segment of the control polygon into $n+1$ pieces, and starting at one end, every nth point is a degree-raised control point.

You should now be able to prove that Bézier curves have linear precision (Exercise 4 of Section 2.3.4).

2.3.3 Functional Bézier Curves

While we are mostly interested in parametric curves in this lecture, the Bernstein–Bézier representation can also be used to represent polynomial functions of the form $y = f(x)$. Suppose we use Bézier control points of the following form: $(i/n, y_i)$, for $i = 0, \ldots, n$. When we weight the control points with the Bernstein polynomials, we get

$$\begin{aligned} B(t) &= \sum_{i=0}^{n} (i/n, y_i) B_i^n(t) \\ &= \left(\sum_{i=0}^{n} (i/n) B_i^n(t), \sum_{i=0}^{n} y_i B_i^n(t) \right) \\ &= \left(t, \sum_{i=0}^{n} y_i B_i^n(t) \right). \end{aligned}$$

Thus, by setting the x-coordinate of the control points equal distances apart, we can represent functions in Bézier form.

2.3.4 Exercises

1. Find the blossom of $F(u) = x^3 + x^2 + x + 1$. Demonstrate that the symmetry property holds by evaluating the blossom at (1, 1, 2), (1, 2, 1), and (2, 1, 1).
2. Find the Bézier control points of $F(x) = x^2 + x^2 + x + 1$.
3. Find the Bézier control points of the quartic representation of $F(x) = x^2 + x^2 + x + 1$ (i.e., degree raise the cubic Bézier curve you constructed in the previous question).
4. Prove that Bézier curves have linear precision.
5. We can evaluate the blossom off the diagonal. Given control points for a cubic polynomial F parameterized over [0, 1],

$$P_0 = (0, 0)$$
$$P_1 = (0, 1)$$
$$P_2 = (1, 1)$$
$$P_3 = (1, 0)$$

   draw a figure for the de Casteljau type of evaluation of f, the blossom of F, at $f(0.25, 0.5, 0.75)$. The symmetry property of the blossom says any order of evaluation yields the same result. Draw this three ways, using the orders (0.25, 0.5, 0.75), (0.5, 0.25, 0.75), and (0.25, 0.75, 0.5).

   Label all the intermediate points with their blossom values, and give the coordinates for all points of evaluation. As an alternative to giving the coordinates of each point of evaluation, you may draw the figures on graph paper (or a grid) with each square being of width 1/8.
6. Suppose we have a parametric cubic curve F in Bézier form and its triaffine blossom f. Let $G(u) = f(1 - u, 1 - u, u)$. G is a polynomial curve. Determine its degree and use blossoming to derive formulae for its Bézier control points over the interval [0, 1].
7. We have seen that a planar parametric cubic Bézier curve may intersect itself (Fig. 2.4). Prove or give a counter example to the following statement: A parametric cubic curve that intersects itself is planar.
8. Suppose we have a planar cubic Bézier curve with control points

$$f(0, 0, 0),\ f(0, 0, 1),\ f(0, 1, 1),\ f(1, 1, 1)$$

   that intersects itself with both points of intersection being in the interval [0, 1]. Prove or give a counter example to the following statement: The control polygon of this Bézier curve intersects itself.

9. Suppose we have quadratic polynomials F and G specified in Bézier form over the interval $[0, 1]$: $F(u) = \sum_{i=0}^{2} F_i B_i^2(u)$ and $G(u) = \sum_{i=0}^{2} G_i B_i^2(u)$. Let $H(u) = (1 - 3u)F(u) + 3uG(u)$. Use blossoming to find the Bézier control points of H over the interval $[0, 1]$. Do not expand the polynomial H.

2.3.5 Implementations

1. Implement an interactive 2D Bézier curve editor with the following functionality:
   - The left mouse button adds a new control point.
   - The middle mouse button is used to move control points.
   - There are two display modes (accessible through a menu):
     - Just the curve.
     - The curve and the control polygon with labeled control points.
   - There should be a reset key/menu option that clears all the control points.

If n control point have been entered, then draw a degree $n - 1$ Bézier curve. This curve should be updated in real time when a control point is moved with the middle mouse button.

2.4 MULTILINEAR BLOSSOM

To discuss the derivatives of a polynomial, it is convenient to work with a second form of the blossom known as the *multilinear blossom*. The multilinear blossom theorem is really just a variation of the affine blossom theorem; since the multilinear blossom theorem gives us derivatives, it would be possible just to state it without proof (or better yet, to state and prove the multilinear blossoming theorem first, after which the multiaffine blossoming theorem follows immediately). However, I have chosen to present the multiaffine blossoming theorem to illustrate the ideas of affine geometry better, and I have chosen to prove both versions of the blossoming theorem because ideas from both proofs are helpful in proving the generalized blossoming theorem (see the exercise of Section 4.1.6).

We begin with a few definitions.

Definition 2.4. *Given* $F(u) = \sum_{i=0}^{n} c_i u^i$, *the corresponding homogeneous polynomial is* $F_*(\bar{u}) = \sum_{i=0}^{n} c_i u^i w^{n-i}$ *where* $\bar{u} = (u, w)$ *for* $w \in \mathcal{R}$.

Note that by setting $w = 1$ we get back F:

$$F_*((u, 1)) = \sum_{i=0}^{n} c_i u^i = F(u)$$

The points $(u, 0)$ correspond to vectors. Also note that each term of F_* is now exactly degree n.

Definition 2.5. *f is multilinear if f is homogeneous and if $f(a_1, \ldots, a_{i-1}, v, a_{i+1}, \ldots, a_n)$ is linear in v.*

If f is multilinear and homogeneous, and if f's variables are univariate, then f only has one term:

$$f(a_1, \ldots, a_n) = c_i \cdot a_1 \cdot \cdots \cdot a_n.$$

In the proof below, we will see what happens if the a_i are multivariate.

We will also need the following form of Taylor expansions.

Definition 2.6. *The Taylor expansion of an analytic function F is given by*

$$F(u) = \sum_{i=0}^{\infty} \frac{F^{(i)}(0)}{i!} u^i$$

If F is a degree n polynomial, this becomes a finite sum with only $n + 1$ terms.

We are now ready to state and prove the multilinear blossoming theorem.

Theorem 2.2. *Theorem:*
Let $F : \mathcal{P} \mapsto \mathcal{Q}$ be a degree n polynomial, where $\mathcal{P}$ and $\mathcal{Q}$ are Euclidean spaces.

1. *There exists a unique map $f_* : (\mathcal{P} \times \mathcal{R})^n \mapsto \mathcal{Q}$ such that*
   (a) f_* is symmetric,
   (b) f_* is multilinear,
   (c) $f_*(\bar{u}, \ldots, \bar{u}) = F(u)$ where $\bar{u} = (u, 1) \in \mathcal{P} \times \mathcal{R}$,
   f_* is the multilinear polar form or multilinear blossom of F.
2. *$f_*(\bar{u}_1, \ldots, \bar{u}_n) = f(u_1, \ldots, u_n)$ where $u_i \in \mathcal{P}$ and $\bar{u}_i = (u_i, 1) \in \mathcal{P} \times \mathcal{R}$*
3. $$F^{(j)}(u) = \frac{n!}{(n-j)!} f_*(\underbrace{\bar{u}, \ldots, \bar{u}}_{n-j}, \underbrace{\delta, \ldots, \delta}_{j})$$

   where $\bar{u} = (u, 1)$ and $\delta = (\vec{1}, 0)$.

Here, $\vec{1}$ is a unit vector in $\mathcal{P}$, and represents a direction for a directional derivative. A nonunit vector may also be used, but then the resulting evaluation of the blossom will need to be rescaled by the length of this vector raised to the jth power. The theorem can be generalized to allow for different δs for each of the j arguments, leading to mixed directional derivatives.

Proof.

Proof of (1): *This is a generalization of the blossoming principle proof. For simplicity, let* $\mathcal{P} = \mathcal{R}$. *Homogenize* $F(u)$ *to* $F_*(\bar{u})$. *Each term of* F_* *is of the form* $c_i u^i w^{n-i}$, *where* $i \leq n$.

Suppose we wish to construct $f_*(\bar{u}_1, \ldots, \bar{u}_n)$ *from* $F_*(\bar{u}) = \sum_{i=0}^{n} c_i u^i w^{n-i}$. *It is easy to convert* F_* *into a* $2n$ *variate function* $f_*^1(\bar{u}_1, \ldots, \bar{u}_n) = f_*^1((u_1, w_1), \ldots, (u_n, w_n))$: *for each term* $c_i u^i w^{n-i}$ *of* F_*, *place a corresponding term* $c_i u_1 \cdot \cdots \cdot u_i \cdot w_{i+1} \cdot \cdots \cdot w_n$ *in* f_*^1. *Clearly,* $f_*^1(\bar{u}, \ldots, \bar{u}) = F_*(\bar{u})$ *if* $\bar{u} = (u, w)$.

But f_*^1 *is not symmetric. Thus, we symmeterize each term of* f_*^1, *converting* $c_i u_1 \cdot \cdots \cdot u_i \cdot w_{i+1} \cdot \cdots \cdot w_n$ *into*

$$\frac{c_i}{\binom{n}{i}} \sum_{\sigma} u_{\sigma(1)} \cdot \cdots \cdot u_{\sigma(i)} \cdot w_{\sigma(i+1)} \cdot \cdots \cdot w_{\sigma(n)}$$

where σ *is a permutation of the integers* $1, \ldots, n$ *with the property that* $\sigma(j) < \sigma(j+1)$ *for* $1 \leq j < i$ *and for* $i < j < n$ *(these restrictions prevent permutations that are equivalent up to permutations of the indices of the* u*s or permutations of the* w*s). The sum of these terms gives us* f_*. *It is readily seen that* f_* *has most of the desired properties, except uniqueness, which is less clear and we prove below. The proof that there are* $\binom{n}{i}$ *permutations for this* σ *is left as an exercise (Exercise 3 of Section 2.4.1).*

Note that (1,c) follows from

$$F(u) = F_*(\bar{u}) = f_*(\bar{u}, \ldots, \bar{u})$$

where $\bar{u} = (u, 1)$.

Observe that f_* *is completely defined by*

$$f_*(\underbrace{\bar{0}, \ldots, \bar{0}}_{n-i}, \underbrace{\delta, \ldots, \delta}_{i}) \quad \forall i = 0, \ldots, n$$

(the proof is similar to that for uniqueness of f *in the multiaffine blossoming theorem). Now we can show uniqueness as follows:*

$$\begin{aligned}
F(u) &= f_*(\bar{u}, \bar{u}, \ldots, \bar{u}) \\
&= f_*(\bar{0} + u\delta, \bar{u}, \ldots, \bar{u}) \\
&= f_*(\bar{0}, \bar{u}, \ldots, \bar{u}) + u f_*(\delta, \bar{u}, \ldots, \bar{u}) \\
&\vdots \\
&= \sum_{i=0}^{n} \binom{n}{i} u^i f_*(\underbrace{\bar{0}, \ldots, \bar{0}}_{n-i}, \underbrace{\delta, \ldots, \delta}_{i}) \\
&= \sum_{i=0}^{n} \binom{n}{i} u^i g_*(\underbrace{\bar{0}, \ldots, \bar{0}}_{n-i}, \underbrace{\delta, \ldots, \delta}_{i}) \\
\Rightarrow f_*(\bar{0}, \ldots, \bar{0}, \delta, \ldots, \delta) &= g_*(\bar{0}, \ldots, \bar{0}, \delta, \ldots, \delta) \quad \textit{as monomials form a basis} \\
\Rightarrow f_* &= g_*
\end{aligned}$$

Proof of (2) *follows from the construction of f and f_*. Consider the construction of a blossom of one term of a monomial:*

$$\begin{array}{cc} F(u) = c_i u^i & F_*(u, w) = c_i u^i w^{n-i} \\ f^1 = c_i u_1 \cdot \cdots \cdot u_i & f_*^1 = c_i u_1 \cdot \cdots \cdot u_i \cdot w_{i+1} \cdot \cdots \cdot w_n \\ f = \frac{c_i}{\binom{n}{i}} \sum_\sigma u_{\sigma(1)} \cdot \cdots \cdot u_{\sigma(i)} & f_* = \frac{c_i}{\binom{n}{i}} \sum_\sigma u_{\sigma(1)} \cdot \cdots \cdot u_{\sigma(i)} w_{\sigma(i+1)} \cdot \cdots \cdot w_{\sigma(n)} \end{array}$$

Both σs give permutations that are equivalent for the u_js, and thus when we set $w_j = 1$ for all j, we have our result.

Proof of (3): *Consider $F(u) = f_*(\bar{u}, \ldots, \bar{u})$ ($\bar{u} = (u, 1) = \bar{0} + u\delta$). From above, we know that*

$$F(u) = \sum_{i=0}^{n} \binom{n}{i} u^i f_*(\underbrace{\bar{0}, \ldots, \bar{0}}_{n-i}, \underbrace{\delta, \ldots, \delta}_{i})$$

But $F(u) = \sum_{i=0}^{n} F^{(i)}(0) u^i / i!$ by Taylor expansion, and since the monomials form a basis, we have

$$F^{(i)}(0) = \frac{n!}{(n-i)!} f_*(\underbrace{\bar{0}, \ldots, \bar{0}}_{n-i}, \underbrace{\delta, \ldots, \delta}_{i})$$

Now, we have the following:

$$\begin{aligned} \frac{n!}{(n-j)!} f_*(\underbrace{\bar{u}, \ldots, \bar{u}}_{n-j}, \underbrace{\delta, \ldots, \delta}_{j}) &= \frac{n!}{(n-j)!} \sum_{k=0}^{n-j} \binom{n-j}{k} f_*(\underbrace{\delta, \ldots, \delta}_{k}, \underbrace{\bar{0}, \ldots, \bar{0}}_{n-j-k}, \underbrace{\delta, \ldots, \delta}_{j}) u^k \\ &= \frac{n!}{(n-j)!} \sum_{k=0}^{n-j} \binom{n-j}{k} f_*(\underbrace{\bar{0}, \ldots, \bar{0}}_{n-j-k}, \underbrace{\delta, \ldots, \delta}_{j+k}) u^k \\ &= \frac{n!}{(n-j)!} \sum_{k=0}^{n-j} \frac{(n-j)!}{k!(n-j-k)!} F^{(j+k)}(0) \frac{(n-j-k)!}{n!} u^k \\ &= \sum_{k=0}^{n-j} \frac{F^{(j+k)}(0)}{k!} u^k \\ &= F^{(j)}(u) \quad \textit{(Taylor expansion)} \end{aligned}$$

Example 2.4. *If $F(u) = 3u^3 + 2u^2 + 6u + 1$ then $F_*(\bar{u}) = 3u^3 + 2u^2 w + 6uw^2 + w^3$ and we construct $f(\bar{u}_1, \bar{u}_2)$ by blossoming each term*:

- $3u^3 \mapsto 3u_1 u_2 u_3$
- $2u^2 w \mapsto 2(u_1 u_2 w_3 + u_2 u_3 w_1 + u_3 u_1 w_2)/3$

- $6uw^2 \mapsto 6(u_1w_2w_3 + u_2w_3w_1 + u_3w_1w_2)/3$
- $w^3 \mapsto w_1w_2w_3$

and

$$\begin{aligned} f_*(\bar{u}_1, \bar{u}_2, \bar{u}_3) = {} & 3u_1u_2u_3 + 2(u_1u_2w_3 + u_2u_3w_1 + u_3u_1w_2)/3 \\ & +2(u_1w_2w_3 + u_2w_3w_1 + u_3w_1w_2) + w_1w_2w_3 \end{aligned}$$

Note that this is multilinear: each term has either u_i or w_i as a linear term. Thus, $\alpha f_*(\bar{u}, \dots) = f_*(\alpha\bar{u}, \dots)$.

Now let us evaluate our multilinear blossom at $f_*(\bar{u}_1, \bar{u}_2, \delta)$. Then

$$\begin{aligned} F(u) &= 3u^3 + 2u^2 + 6u + 1 \\ f_*(\bar{u}_1, \bar{u}_2, \bar{u}_3) &= 3u_1u_2u_3 + 2(u_1u_2w_3 + u_2u_3w_1 + u_3u_1w_2)/3 \\ &\quad +2(u_1w_2w_3 + u_2w_3w_1 + u_3w_1w_2) + w_1w_2w_3 \\ f_*(\bar{u}_1, \bar{u}_2, \delta) &= 3u_1u_2 + 2(0 + u_2w_1 + u_1w_2)/3 + 2(0 + 0 + w_1w_2) + 0 \\ &= 3u_1u_2 + 2(u_2w_1 + u_1w_2)/3 + 2w_1w_2 \\ f_*(\bar{u}, \bar{u}, \delta) &= 3u^2 + 4u/3 + 2 \\ &= F^{(1)}(u)/3 \end{aligned}$$

By computing the derivative of F in the usual fashion, we see that the last step is true. While the above is an awkward way to compute the derivative of a polynomial (especially if it is in monomial form), it is only intended as an example to show that the math really does work. In practice, since we usually work in a Bézier representation, we compute the derivative as a scaled differences of control points.

We can go either way to get the multilinear blossom: homogenize F to get F_*, and then blossom F_* to get f_*, or blossom F to get f, and then homogenize f to get f_*. Likewise, we can go back in either direction.

2.4.1 Exercises

1. Homogenize $F(x) = x^3 + x^2 + x + 1$.
2. Find the multilinear linear blossom f_* of $F(x) = x^3 + x^2 + x + 1$. Evaluate $F'(1)$ by taking the derivative of F directly then evaluating at 1, and by evaluating the blossom at $f_*(1, 1, \delta)$.
3. Prove that the σ used in the proof the the multilinear blossom theorem has $\binom{n}{i}$ permutations.

2.5 DERIVATIVES OF BÉZIER CURVES

Now that we have the multilinear blossom, we can describe the derivatives of Bézier curves. We know the derivatives of F in terms of the multilinear blossom are

$$F^{(j)}(u) = \frac{n!}{(n-j)!} f_*(\underbrace{\bar{u}, \ldots, \bar{u}}_{n-j}, \underbrace{\delta, \ldots, \delta}_{j})$$

where $\bar{u} = (u, 1)$ and $\delta = (1, 0)$. What does this mean in terms of the control points of F?

We first introduce the notation $\bar{1}^{\langle k \rangle}$, which is similar to the notation Farin uses [7]. It means replicate the argument "$\bar{1}$" k times. In this notation, the previous equation is written as

$$F^{(j)}(u) = \frac{n!}{(n-j)!} f_*(\bar{u}^{\langle n-j \rangle}, \delta^{\langle j \rangle})$$

Consider now the first derivative of F parameterized over the interval $[0, 1]$:

$$F'(u) = n f_*(\bar{u}^{\langle n-1 \rangle}, \delta)$$

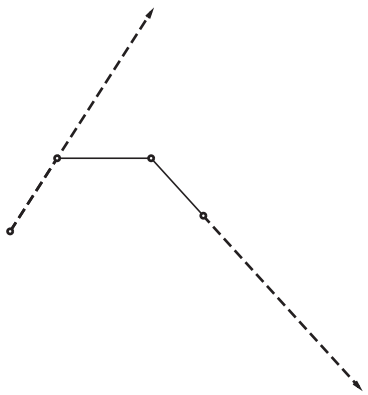
We can write $\delta = (1, 0)$ as $\delta = (1, 1) - (0, 1)$, allowing us to rewrite the first derivative of F:

$$\begin{aligned} F'(u) &= n f_*(\bar{u}^{\langle n-1 \rangle}, \delta) \\ &= n f_*(\bar{u}^{\langle n-1 \rangle}, (1, 1) - (0, 1)) \\ &= n(f_*(\bar{u}^{\langle n-1 \rangle}, (1, 1)) - f_*(\bar{u}^{\langle n-1 \rangle}, (0, 1))) \\ &= n(f(u^{\langle n-1 \rangle}, 1) - f(u^{\langle n-1 \rangle}, 0)) \end{aligned}$$

In particular, the first derivative of F at 0 is

$$F'(0) = n(f(0^{\langle n-1 \rangle}, 1) - f(0^{\langle n-1 \rangle}, 0)),$$

which is just an integer multiple of the difference between the first two control points of F:



If F is parameterized over an arbitrary interval $[a, b]$, then we have to be a little trickier:

$$\begin{aligned}
F'(u) &= n f_*\big(\bar{u}^{\langle n-1\rangle}, \delta\big) \\
&= n f_*\left(\bar{u}^{\langle n-1\rangle}, \frac{b-a}{b-a}\delta\right) \\
&= \frac{n}{b-a} f_*\big(\bar{u}^{\langle n-1\rangle}, (b,1) - (a,1)\big) \\
&= \frac{n}{b-a}\big(f_*\big(\bar{u}^{\langle n-1\rangle}, (b,1)\big) - f_*\big(\bar{u}^{\langle n-1\rangle}, (a,1)\big)\big) \\
&= \frac{n}{b-a}\big(f\big(u^{\langle n-1\rangle}, b\big) - f\big(u^{\langle n-1\rangle}, a\big)\big) \\
F'(a) &= \frac{n}{b-a}\big(f\big(a^{\langle n-1\rangle}, b\big) - f\big(a^{\langle n-1\rangle}, a\big)\big)
\end{aligned}$$

For higher order derivatives, we get one factor of $1/(b-a)$ for each derivative.

Referring back to Fig. 2.6, we see that to take a derivative at an arbitrary location t on the curve, we can evaluate with de Casteljau's algorithm, and apply this formula to either the interval $[0, t]$ or $[t, 1]$. Thus, we see that the de Casteljau algorithm gives us both the position and first derivative of a curve at a parameter value t (in fact, it gives us all the derivatives at t).

What about the derivative of $F(u)$ at arbitrary u?

$$\begin{aligned}
\frac{F'(u)}{n} &= f_*\big(\bar{u}^{\langle n-1\rangle}, \delta\big) \\
&= u f_*\big(\bar{1}, \bar{u}^{\langle n-2\rangle}, \delta\big) + (1-u) f_*\big(\bar{0}, \bar{u}^{\langle n-2\rangle}, \delta\big) \\
&\vdots \\
&= \sum_{i=0}^{n-1} \binom{n-1}{i} u^i (1-u)^{n-i-1} f_*\big(\bar{0}^{\langle n-i-1\rangle}, \bar{1}^{\langle i\rangle}, \delta\big) \\
&= \sum_{i=0}^{n-1} \binom{n-1}{i} u^i (1-u)^{n-i-1} \big[f_*\big(\bar{0}^{\langle n-i-1\rangle}, \bar{1}^{\langle i\rangle}, \bar{1}\big) - f_*\big(\bar{0}^{\langle n-i-1\rangle}, \bar{1}^{\langle i\rangle}, \bar{0}\big)\big] \\
&= \sum_{i=0}^{n-1} \big[f_*\big(\bar{0}^{\langle n-i-1\rangle}, \bar{1}^{\langle i\rangle}, \bar{1}\big) - f_*\big(\bar{0}^{\langle n-i-1\rangle}, \bar{1}^{\langle i\rangle}, \bar{0}\big)\big] B_i^{n-1}(u)
\end{aligned}$$

Thus, the first derivative of F is itself a Bézier "curve" of one lower degree. Here, the control "points" are vectors rather than points. Thus, the second derivative can either be obtained from the original derivative formula, or it can be calculated by taking the derivative of the first derivative formula.

2.5.1 Exercises

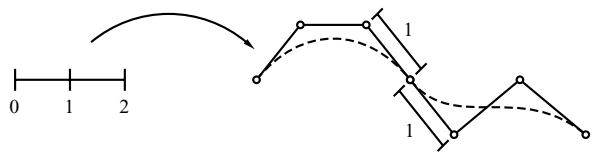
1. Use the multiaffine blossom of $F(x) = x^3 + x^2 + x + 1$ to find $F'(0)$.
2. If we repeat the first control point of a Bézier curve B (parameterized over $[0, 1]$) then $B'(0) = 0$. However, the tangent line to the curve at this point exists. Find the tangent line to $B(0)$ for the portion of the curve parameterized over $[0, 1]$. Prove your result.
3. Given a two-space quadratic polynomial in Bézier form over the interval $[0, 1]$ (this specifies the control points; the domain of the curve is the entire real line) and its biaffine blossom f, is there a blossom value of f for every point in the range? If so, give a formula/algorithm for determining a range point's blossom arguments. If not, state which regions of the plane have blossom values and which do not, and give a formula/algorithm for the pre-image of every point in the valid region; i.e., given a point (x, y) in the plane, find u, v such that $f(u, v) = (x, y)$.

2.6 CONTINUITY

Now that we know how to compute derivatives, we can consider joining two curves together "smoothly." We begin with a brief review of continuity. Two curves $F(t)$ and $G(t)$ are said to meet with C^k continuity at t_0 if $F^{(i)}(t_0) = G^{(i)}(t_0)$ for $0 \leq i \leq k$. If there is a discontinuity in the position of a curve at parameter value t_0, then in geometric modeling, we commonly say that the curve is "C^{-1}" at t_0. A curve is said to be C^∞ if all its derivatives are continuous everywhere.

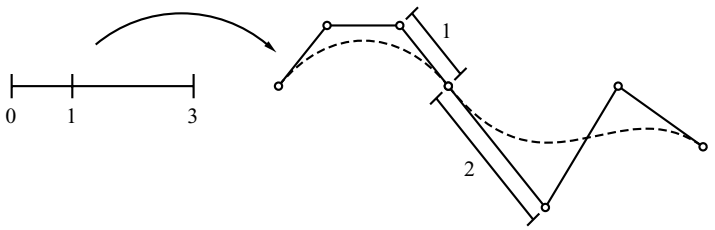
In mathematics, a "smooth" curve usually refers to a C^∞ curve However, in geometric modeling, a "smooth" curve usually refers to a piecewise C^∞ curve, where the pieces meet with at least equal position and first derivatives. We will begin by looking at what is required for two Bézier curves to meet with continuous position and first derivatives (or C^1 continuity).

Suppose we wish to join two Bézier curves together with continuous position and first derivatives. How do we do this? C^0 continuity is trivial: you set the first control point of the second curve equal to the last control point of the first curve. Then, assuming both curves have the same length parameterizations (e.g., $[0, 1]$ and $[1, 2]$), the results on the first derivative shows us how to get C^1 continuity between the two curves:



In this figure, the line segments labeled with "1" indicate that the length of the last segment of the control polygon of the first curve is equal to the first segment of the control polygon of the second segment.

What if we want to join two curve segments with different parameterizations (e.g., [0, 1] and [1, 3])? The condition is similar, but rather than first have first/last segments of the control polygons be of equal length, they instead have to have lengths in the same ratios as the lengths of their domain segments:



2.6.1 Cubic Hermite Interpolation

Given points $P_0, \ldots, P_L$, vectors $\vec{v}_0, \ldots, \vec{v}_L$, and a uniform knot vector $0, \ldots, L$, find a piecewise cubic function H consisting of L cubic curves segments H_i, such that for $i = 0, \ldots, L-1$ the following hold:

- $H_i(i) = P_i$
- $H_i(i+1) = P_{i+1}$
- $H_i'(i) = \vec{v}_i$
- $H_i'(i+1) = \vec{v}_{i+1}$

You can find blending functions for the Ps and $\vec{v}$s. Alternatively, you can write this in Bézier form. If we let P_0^i, P_1^i, P_2^i, and P_3^i be the Bézier control points for H_i, then for $i = 0, \ldots, L-1$, we set these points to the following:

- $P_0^i = P_i$
- $P_3^i = P_{i+1}$
- $P_1^i = P_i + \vec{v}_i/3$
- $P_2^i = P_{i+1} - \vec{v}_{i+1}/3$

2.6.2 C^1 Continuity and the Blossom

What does C^1 continuity say about the blossoms of f and g? First, we have $f(1,1,1) = g(1,1,1)$ for C^0 continuity. Next, we know that $f(1,1,1) - f(1,1,0) = g(1,1,2) -$

$g(1, 1, 1)$, or alternatively, $f_*(\bar{1}, \bar{1}, \delta) = g_*(\bar{1}, \bar{1}, \delta)$. This gives us the following:

$$\begin{aligned} f_*(\bar{1}, \bar{1}, \delta) &= g_*(\bar{1}, \bar{1}, \delta) \\ f_*(\bar{1}, \bar{1}, (u-1)\delta) &= g_*(\bar{1}, \bar{1}, (u-1)\delta) \\ f_*(\bar{1}, \bar{1}, \bar{u}) - f_*(\bar{1}, \bar{1}, \bar{1}) &= g_*(\bar{1}, \bar{1}, \bar{u}) - g_*(\bar{1}, \bar{1}, \bar{1}) \\ f_*(\bar{1}, \bar{1}, \bar{u}) &= g_*(\bar{1}, \bar{1}, \bar{u}) \\ f(1, 1, u) &= g(1, 1, u) \end{aligned}$$

Thus, if f and g meet with C^1 continuity at $u = 1$, then the above equation holds. And the converse is clearly true: if the above equation holds for all u, then f and g meet with C^1 continuity.

We can also prove the above without resorting to the multilinear blossom:

$$\begin{aligned} f(1, 1, 1) - f(1, 1, 0) &= g(1, 1, 2) - g(1, 1, 1) \\ f(1, 1, 0) &= g(1, 1, 1) - g(1, 1, 2) + f(1, 1, 1) \\ &= g(1, 1, 1) - g(1, 1, 2) + g(1, 1, 1) \\ &= g(1, 1, 0) \\ f(1, 1, u) &= (1-u)f(1, 1, 0) + uf(1, 1, 1) \\ &= (1-u)g(1, 1, 0) + ug(1, 1, 1) \\ &= g(1, 1, u) \end{aligned}$$

Further, we can use this as a test for continuity: if we want to know if two Bézier segments meet with C^1 continuity (assuming they meet C^0), we extend the first/last segments and see if they have the same blossom values (we can also do a ratios test based on the relative lengths of the domains).

In general, we know that for f and g to meet with C^k continuity at u, then for $i = 0 \ldots k$, the following must hold:

$$f_*(\bar{u}^{\langle n-i \rangle}, \delta^{\langle i \rangle}) = g_*(\bar{u}^{\langle n-i \rangle}, \delta^{\langle i \rangle})$$

Because f_* and g_* are multilinear, we can show the following theorem using a proof by induction.

Theorem 2.3. *For F and G to meet with C^k continuity at u, we must have*

$$f_*(\bar{u}^{\langle n-i \rangle}, \bar{u}_1, \ldots, \bar{u}_i) = g_*(\bar{u}^{\langle n-i \rangle}, \bar{u}_1, \ldots, \bar{u}_i),$$

for $i = 0 \ldots k$. Note that it is sufficient to show that

$$f_*(\bar{u}^{\langle n-k \rangle}, \bar{u}_1, \ldots, \bar{u}_k) = g_*(\bar{u}^{\langle n-k \rangle}, \bar{u}_1, \ldots, \bar{u}_k),$$

Proof. Base case: For $\ell = 0$ we already know that

$$f_*(\bar{u}^{\langle n-k \rangle}, \delta^{\langle k \rangle}) = g_*(\bar{u}^{\langle n-k \rangle}, \delta^{\langle k \rangle}).$$

Assume that we know

$$f_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \delta^{\langle k-\ell\rangle}) = g_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \delta^{\langle k-\ell\rangle})$$

Then

$$\begin{aligned}
f_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \delta^{\langle k-\ell\rangle}) &= g_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \delta^{\langle k-\ell\rangle}) \\
(u_{\ell+1} - u) f_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \delta^{\langle k-\ell\rangle}) &= (u_{\ell+1} - u) g_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \delta^{\langle k-\ell\rangle}) \\
f_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, (u_{\ell+1} - u)\delta, \delta^{\langle k-\ell-1\rangle}) &= g_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, (u_{\ell+1} - u)\delta, \delta^{\langle k-\ell-1\rangle}) \\
f_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \bar{u}_{\ell+1} - \bar{u}, \delta^{\langle k-\ell-1\rangle}) &= g_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \bar{u}_{\ell+1} - \bar{u}, \delta^{\langle k-\ell-1\rangle}) \\
f_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \bar{u}_{\ell+1}, \delta^{\langle k-\ell-1\rangle}) - & \\
f_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \bar{u}, \delta^{\langle k-\ell-1\rangle}) &= g_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \bar{u}_{\ell+1}, \delta^{\langle k-\ell-1\rangle}) \\
&\quad - g_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_\ell, \bar{u}, \delta^{\langle k-\ell-1\rangle}) \\
f_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_{\ell+1}, \delta^{\langle k-\ell-1\rangle}) &= g_*(\bar{u}^{\langle n-k\rangle}, \bar{u}_1, \ldots, \bar{u}_{\ell+1}, \delta^{\langle k-\ell-1\rangle})
\end{aligned}$$

concluding the inductive proof.

And now from the multilinear blossom theorem, we get

$$f(u^{\langle n-i\rangle}, u_1, \ldots, u_i) = g(u^{\langle n-i\rangle}, u_1, \ldots, u_i)$$

2.6.3 C^2 Continuity

If we have two cubic curves F and G meeting with C^2 continuity at u, what are the geometric constraints on the control points of F and G?

Suppose F is parametrized over $[r, s]$ and G is parametrized over $[s, t]$. We know $f(s, s, s) = g(s, s, s)$, and we know the conditions on $f(r, s, s)$ and $g(s, s, t)$. Consider the two lines defined by $f(r, r, s)$ and $f(r, s, s)$ and by $g(s, t, t)$ and $g(s, s, t)$. We can write these two lines as $f(r, s, u)$ and $g(s, t, v)$, where u and v are free parameters. In general, these two lines will intersect in a single point. But, since f and g meet C^2, they agree whenever at least one of their parameters is s. Thus, the point of intersection must be $f(r, s, t) = g(r, s, t)$ (see Fig. 2.8).

Further, we see from the figure that certain ratios must exist along various lines. This arrangement of control points is known as an *A-frame* construction, since you can see an "A" (drawn in bold but a bit skew in this figure, from $f(r, r, s)$ to $f(r, s, t)$ to $f(s, t, t)$, with the crossbar of "A" running from $f(r, s, s)$ to $f(s, s, t)$).

2.6.4 C^k Continuity

In general, we have C^k continuity between degree n polynomials F and G at t if we know f and g are equal on $k + 1$ sets of blossom arguments having $n - k$ arguments set to t. Some restrictions apply to these sets. However, the idea is that we must choose these $k + 1$ sets of

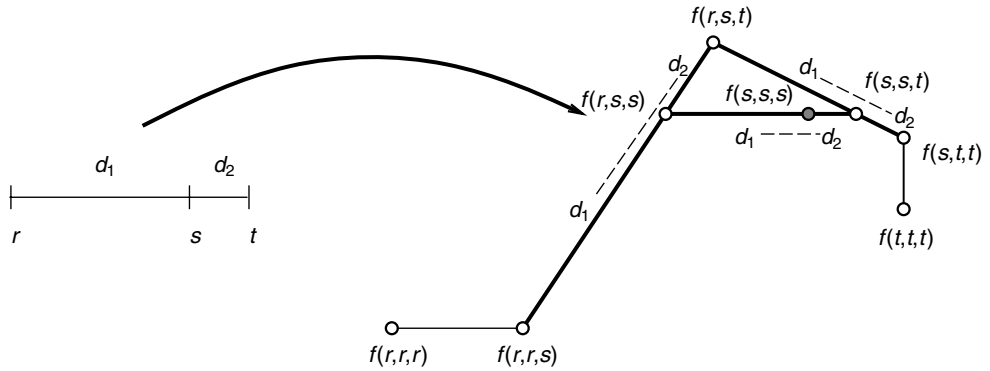


FIGURE 2.8: C^2 continuity between Bézier segments

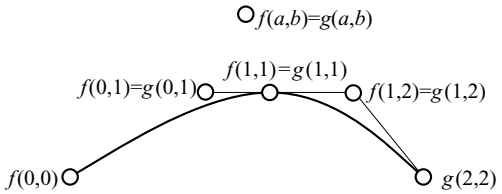
arguments so that we can construct $f(t^{\langle n-k\rangle}, t_1, \ldots, t_k)$ for all t_i. In particular, the $k+1$ Bézier points of F (or G) that are closest to t will suffice. Thus, if we consider the portion of F over $[s, t]$ and we wish it to meet G with C^k continuity at t, then f and g must agree at $f(t^{\langle n\rangle})$, $f(s, t^{\langle n-1\rangle}), \ldots, f(s^{\langle k\rangle}, t^{\langle n-k\rangle})$.

2.6.5 Exercises

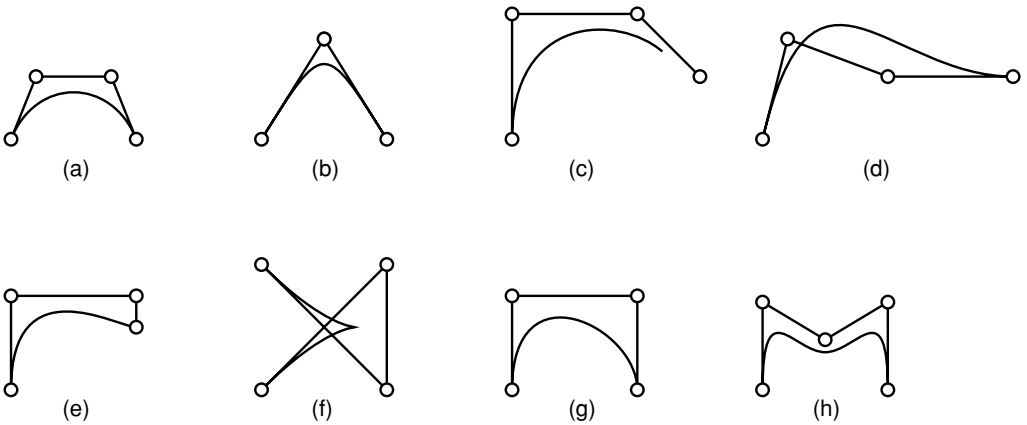
1. In Section 2.6.1, we considered the cubic Hermite interpolation problem when the knots were consecutive integers. Here we will generalize the problem to arbitrary knot vectors. Given points $P_0, \ldots, P_L$, vectors $\vec{v}_0, \ldots, \vec{v}_L$, and a knot vector $t_0 < \cdots < t_L$, find the cubic Bézier control points for the cubic Hermite curve that interpolates this data, i.e., H is this cubic Hermite curve if
   - $H(t_i) = P_i$,
   - $H'(t_i) = \vec{v}_i$.
2. Draw the diagram for two 4th-degree Bézier curves meeting with C^3 continuity. Assume that the first curve is parametrized over $[0, 1]$ and the second over $[1, 2]$.
3. Give the conditions for two curves to meet with C^3 continuity. Draw a picture. Hint: Start with a degree 4 curve, subdivide it, and think about what the condition will be.
4. Suppose we have two quadratic Bézier curves, F parametrized over $[0, 1]$ and G parametrized over $[1, 2]$. Further, suppose that F and G meet C^1 at 1. If $F(t) = G(t)$ for some $t \neq 1$, then F and G are merely different representations of the same curve.

   Instead consider the blossoms of F and G. Since F and G meet C^1 at 1, we know that $f(1, 1) = g(1, 1)$ and $f(1, u) = g(1, u)$ for all u. If we knew that $f(t, t) = g(t, t)$ for some $t \neq 1$, then again we would know that $f = g$ as this is the situation described in the previous paragraph. Prove further that if we know $f(a, b) = g(a, b)$ for some

arbitrary a and b with $a \neq b$, $a \neq 1$, and $b \neq 1$, then f and g are identical functions (and thus, $F = G$):



5. Below are eight curves and their "control points." Two sets of control points are the Bézier control points for the corresponding curve. The other six are not. Determine which two are the Bézier points for the curve, and for the other six, give a reason that they are *not* Bézier control points. Assume that no control point is duplicated:



2.7 CHANGE OF BASIS

In Sections 2.2 and 2.3.1, we saw triangle diagrams as a means of illustrating and analyzing de Casteljau's algorithm. Another of the many uses for triangle diagrams is changing polynomial bases, for example, changing from the monomial representation to the Bernstein representation. We have already seen a change of basis between Bézier representations via de Casteljau's algorithm: given a Bézier curve parameterized over the interval $[s, t]$, if we perform a de Casteljau evaluation at u, then the edges of the triangle diagram give us the Bézier control points for the same polynomial over the intervals $[s, u]$ and $[u, t]$ (see Section 2.3.1).

We now look at performing a change of basis from the monomial basis to the Bernstein basis. In the proof of Theorem 2.2, part (1), we saw that the monomial representation of a

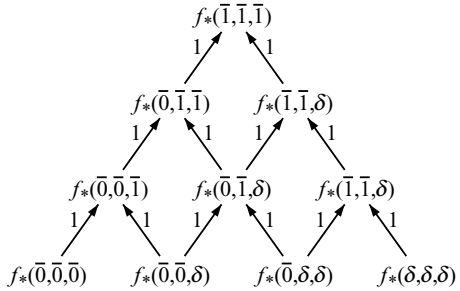


FIGURE 2.9: Change of basis from monomial to Bernstein basis

polynomial F, $F(u) = \sum_0^n c_i u^i$, has its coefficients c_i related to the blossom of F as follows:

$$c_i = \binom{n}{i} f_*(\bar{0}^{\langle n-i\rangle}, \delta^{\langle i\rangle})$$

For example, for a cubic curve parameterized over the interval $[0, 1]$, the scaled monomial control points are

$$f_*(\bar{0}, \bar{0}, \bar{0}),\ f_*(\bar{0}, \bar{0}, \delta),\ f_*(\bar{0}, \delta, \delta),\ f_*(\delta, \delta, \delta)$$

(Since the argument δ represents a vector rather than a point in the domain, we must use the homogeneous form of the blossom.) Evaluating the polynomial in this form using a triangle diagram is a bit different than when it is in the Bernstein/Bézier form; in particular, the coefficients along two edges leading into a node no longer have to sum to 1. See Exercise 3 of Section 2.8.

We can convert from monomial to Bernstein basis as illustrated in Fig. 2.9. The Bézier control points appear along the left edge of the triangle diagram.

2.8 EXERCISES

1. Draw the triangle diagram explicitly converting the coefficients of the monomial $x^3 + x^2 + x + 1$ to Bernstein/Bézier form.
2. Draw a triangle diagram for converting from cubic Bézier form to the (scaled) monomial basis.
3. Draw a triangle diagram for evaluating a cubic scaled monomial $F(u)$ with blossom coefficients $f_*(\bar{0}, \bar{0}, \bar{0}),\ f_*(\bar{0}, \bar{0}, \delta),\ f_*(\bar{0}, \delta, \delta),\ f_*(\delta, \delta, \delta)$ at t.

2.9 FAST EVALUATION

How do we evaluate a curve quickly? For univariate polynomials, if we evaluate a degree n polynomial in monomial form by evaluating x^i for each i and multiplying by c_i, it takes n additions and $n(n+1)/2$ multiplications for each dimension of our range.

When speed is a concern, Horner's rule is the common technique for fast evaluation:

$$\begin{aligned} p(x) &= a + bx + cx^2 + dx^3 \\ &= a + x(b + x(c + dx)) \end{aligned}$$

Thus, each evaluation requires only n additions and n multiplications for each dimension of the range.

However, suppose we want to evaluate our polynomial at a sequence of points uniformly sampled over our domain. In this situation, *forward differencing* is a technique that is even faster than the Horner's rule.

Suppose we have a polynomial $F(u)$ in Bernstein–Bézier form defined over the interval $[0, 1]$. (Since we are in Bernstein–Bézier form, unless we care about the parameter value of each sample point, we can consider the curve to be parameterized over $[0, 1]$ regardless of its actual parameterization.) We first convert to the monomial representation of the multilinear blossom of F with control points $f_*(\bar{0}^{\langle n \rangle}), f_*(\bar{0}^{\langle n-1 \rangle}, \delta), \ldots, f_*(\delta^{\langle n \rangle})$ (see Section 2.7). Our goal is to compute $F(0 + ih)$ for i running from 0 to S, where $S + 1$ is the number of samples of F that we want and $h = 1/S$.

Starting with a linear function $F(u)$, consider computing $F(u + h)$ when we already know $F(u)$:

$$\begin{aligned} F(u + h) = f(u + h) &= f_*(\bar{u} + h\delta) \\ &= f_*(\bar{u}) + hf_*(\delta) \\ &= f(u) + hf_*(\delta) \\ &= F(u) + \Delta_1, \end{aligned}$$

where $\Delta_1 = hf_*(\delta)$. If we precompute $\Delta_1 = hf_*(\delta)$, then we can compute $f(u + h)$ from $f(u)$ with a single addition (albeit an addition of an n-dimensional point with an n-dimensional vector) and in turn we can compute $f(u + 2h)$ from $f(u + h)$ with a second addition reusing the same precomputed Δ_1. Since we want to start at $u = 0$, the following code will compute the values we want

```
u = 0
F[0] = f(0)
Δ1 = hf*(δ)
for i = 1 to S
    u = u + h
    F[i] = F[i − 1] + Δ1
end
```

At the end of this code sequence, we will have samples of $F(u)$ stored in an array $F[i]$; the line of code $u = u + h$ is not needed unless we also want the domain values associated with the points of evaluation.

Now suppose $F(u)$ is quadratic. Then

$$\begin{aligned} F(u+h) &= f(u+h, u+h) = f_*(\bar{u}+h\delta, \bar{u}+h\delta) \\ &= f_*(\bar{u}, \bar{u}+h\delta) + hf_*(\delta, \bar{u}+h\delta) \\ &= f_*(\bar{u}, \bar{u}) + 2hf_*(\bar{u}, \delta) + h^2 f_*(\delta, \delta) \\ &= F(\bar{u}) + \Delta_1(u) \end{aligned}$$

where $\Delta_1(u) = 2hf_*(\bar{u}, \delta) + h^2 f_*(\delta, \delta)$. However, unlike in the linear case, $\Delta_1(u)$ is a function of u. Thus, precomputing it once will not allow us to repeatedly add it to $F(u + ih)$ to obtain $F(u + (i+1)h)$ unless we update Δ_1 at each step. Since $\Delta_1(u)$ is a linear function, we can find a forward differencing step to update it:

$$\begin{aligned} \Delta_1(u+h) &= 2hf_*(\bar{u}+h\delta, \delta) + h^2 f_*(\delta, \delta) \\ &= 2h[f_*(\bar{u}, \delta) + hf_*(\delta, \delta)] + h^2 f_*(\delta, \delta) \\ &= \Delta_1(u) + 2h^2 f_*(\delta, \delta) \\ &= \Delta_1(u) + \Delta_2 \end{aligned}$$

where $\Delta_2 = 2h^2 f_*(\delta, \delta)$.

This leads to the following code:

```
u = 0
F[0] = f(0, 0)
Δ1 = 2hf*(0̄, δ) + h²f*(δ, δ)
Δ2 = 2h²f*(δ, δ)
for i = 1 to S
    u = u + h
    F[i] = F[i − 1] + Δ1
    Δ1 = Δ1 + Δ2
end
```

and we see that we compute each evaluation of F with two additions (again, the line of code $u = u + h$ is not needed unless we want the domain values).

In general, by using forward differencing, each evaluation of the curve will take n additions (albeit multidimensional additions) and no multiplications (other than the multiplications in the initialization steps).

de Casteljau's algorithm, on the other hand, takes $n(n+1)/2$ affine combinations, each of which requires two multiplications and one addition. Why would we consider using such a slow algorithm? The main reason is that the code for de Casteljau's algorithm is simpler to write than is the code for forward differencing, especially if you want arbitrary degrees. While there are other factors that might seem to favor de Casteljau's algorithm, they are of lessor importance. For example, in addition to evaluating for position, we also get the derivatives of the polynomial when we use de Castlejau's algorithm. This is not a major consideration, since both Horner's rule and forward differencing can also give you the derivatives at a lower cost than de Castlejau's algorithm. Another observation is that de Castlejau's algorithm is numerically more stable stable than is forward differencing. However, for rendering purposes, evaluating a cubic curve with forward differencing is easily stable enough to draw a polynomial curve, especially if double precision floating point numbers are used.

And finally, while speed is the primary advantage of forward differencing, realize that de Casteljau's algorithm is easily fast enough to draw curves at interactive rates. Thus, the complicated fast forward differencing algorithm is unlikely to be chosen over de Casteljau's algorithm for drawing curves. However, we will briefly return to forward differencing when we look at surfaces, where the additional number of evaluation points may again make forward differencing an algorithm worth considering.

2.9.1 Exercise

1. Derive a forward difference algorithm for cubic polynomials and give pseudo-code that implements it.

2.9.2 Implementations

1. Implement code for a forward differencing algorithm for cubic Bézier curves. Draw your curve using both forward differencing and de Casteljau's algorithm to verify the correctness of the algorithm and to convince yourself that forward differencing has reasonable numerical stability properties.

   Do your computations with floats (instead of doubles) and see what value of S is required for the forward differencing computed curve to diverge from the de Casteljau algorithm computed curve.

CHAPTER 3

B-Splines

Suppose we want to draw a complicated curve. We could use a high-degree Bézier curve, but there are several problems with this approach:

- We cannot represent cusps (well, not easily, anyway).
- It is expensive to evaluate high-degree Bézier curves.
- A large number of control points may be required to get the shape we want.
- The effect of modifying middle control points is diffuse, and they become less useful as shape parameters.

For these reasons, piecewise polynomial curves are used instead.

What we would like is a piecewise polynomial curve technique where the segments automatically join with a high level of continuity. By "automatic," we mean that we do not want to have to constrain the location where users can place any of the control points to get the desired level of continuity. And ideally, if we have a degree n curve, then we could automatically have C^{n-1} continuity. To find such a curve, we will look at smoothly joined Bézier curves and try to reduce the number of points needed to represent them.

Consider a C^1 piecewise cubic curve represented in Bézier form. If the curve has L segments, then naively we would store $4L$ control points. But since the first control point of segment i is equal to the last control point of segment $i-1$, we really only need storage for $3L+1$ control points (plus, possibly, storage for a knot vector $\{t_0, \dots\}$). Now consider the junction between two of the segments, say $f_i(s,s,s) = f_{i+1}(s,s,s)$. That control point is redundant, since we can compute it from the control points on either side. That is, if f_i is parameterized over $[r,s]$ and f_{i+1} is parameterized over $[s,t]$, then

$$\begin{aligned}\frac{(t-s)}{t-r} f_i(r,s,s) + \frac{(s-r)}{t-r} f_{i+1}(s,s,t) &= f\Big(\frac{(rt-rs)+(ts-tr)}{t-r}, s, s\Big) \\ &= f_i(s,s,s)\end{aligned}$$

Thus, we would only need to store $2L+2$ control points (Fig. 3.1). Likewise, if we have a C^2 piecewise cubic, then we would only need to store $L+3$ control points (Fig. 3.2), since as

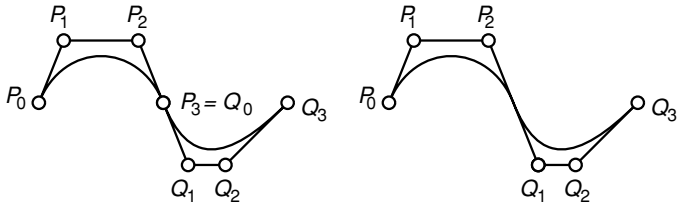


FIGURE 3.1: Reducing data used to represent C^1 joined cubic Bézier segments

shown for two cubic with uniform knots in the figure, the missing Bézier control points between two gray points are located at 1/3 and 2/3 away along the segment between the gray points (the ends of the curve are special cases, where the missing Bézier point lies half-way between the black and the gray points). Further, if we look at the blossom values of the gray points (see Fig. 2.8), we find that the blossom values associated with the gray points have arguments that are successive knot values.

Alternatively, we can go the other way: start with a knot sequence t_i and a set of points P_i, and set these points to be the blossom values $f(t_i, t_{i+1}, t_{i+2})$. If we now solve for the corresponding Bézier points (i.e., moving from right to left in Fig. 3.2), then we know the piecewise cubic curve is C^2 *by construction.*

Note that I have left off some details: how many knots? How many control points? What about end conditions? But the idea should be clear: we can construct a C^2 piecewise, cubic curve by specifying roughly one point per cubic segment, with no constraints on the control points.

The result is a cubic B-spline curve, with the gray points in the right subfigure of Fig. 3.2 being the cubic B-spline control points.

More formally, we can define a B-spline curve as follows:

Definition 3.1. *Given a knot vector* $t_0 \le t_1 \le \cdots \le t_{L+2n-2}$, *data points* $p_0, \ldots, p_{L+n-1}$. *Then, for* $0 \le i \le L-1$ *the* i*th interval of a degree* n *B-spline curve* F *is defined over* $[t_{i+n-1}, t_{i+n}]$ *by the* $n+1$ *values of its blossom*

$$f(t_k, \ldots, t_{k+n-1}) = p_k$$

for $k = i, \ldots, i+n$.

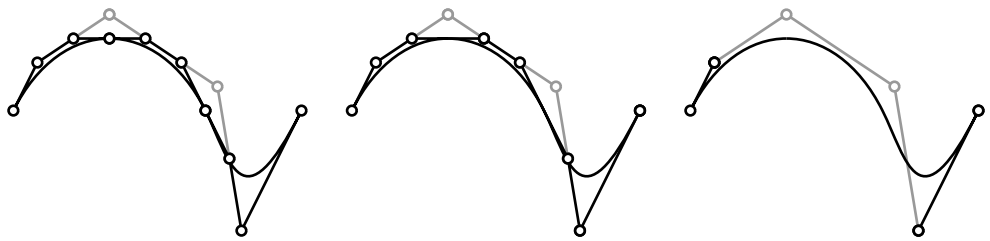
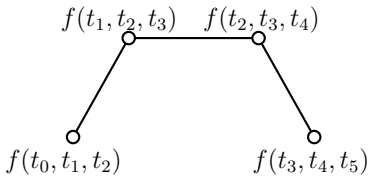


FIGURE 3.2: Reducing data used to represent C^2 joined cubic Bézier segments

If a knot is not unique (i.e., $t_i = t_{i+1}$) then we count the number of times the knot occurs and call that the *multiplicity* of the knot. If the multiplicity equals the degree, we say the knot has full multiplicity for reasons we will see shortly. For now, assume all knots have multiplicity one.

Note that we have n more control points than intervals, and $2n - 1$ more knot values than intervals. (Some definitions use two more knot values for reasons we will see on page 53.)

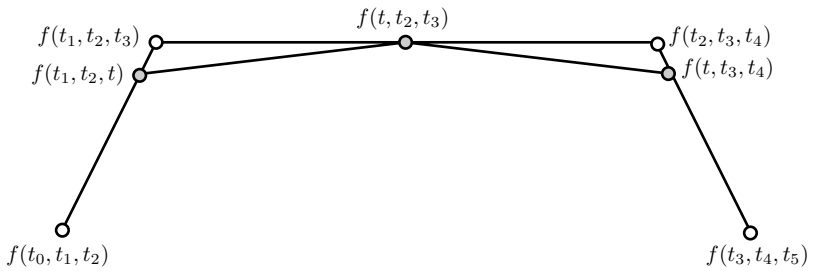
If you do not want to memorize the above formulas, you can reconstruct them from a blossom diagram. The following is a diagram for a single B-spline segment with the knot vector $\{t_0, t_1, t_2, t_3, t_4, t_5\}$:



From this diagram, we can obtain the relationship between the number of control points and the number of knots, and which knots are used as the blossom arguments for each control point. Further, although a single segment of a B-spline curve can be evaluated at any value in the domain, when considering a piecewise polynomial curve, we restrict the domain to a subinterval. That subinterval starts at the value of the last knot argument of the first control and runs to the first knot argument of the last control point (e.g., $[t_2, t_3]$ in the figure above).

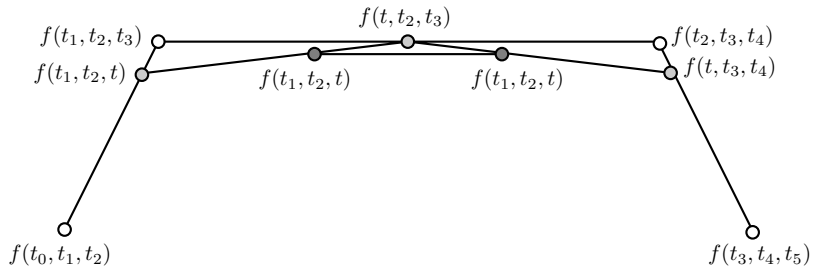
Given that we have a B-spline curve, we can ask "how do we evaluate the curve?" We could convert it to Bézier form and evaluate the appropriate Bézier segment. However, there are also several other methods. We will look at the one based on repeated knot insertion.

Suppose we have a B-spline curve as defined above, and we wish to insert a new knot to get a new knot vector and a new set of control points for the same curve. The new knot vector we know (it is just the old one with the new knot inserted). Blossoming shows us how to find the new control points for the B-spline with the knot vector $\{t_0, t_1, t_2, t, t_3, t_4, t_5\}$:

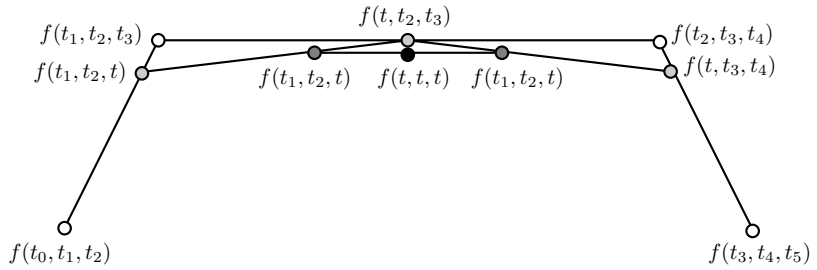


Thus, starting with the original set of control points, we remove $f(t_1, t_2, t_3)$ and $f(t_2, t_3, t_4)$ and add $f(t_1, t_2, t)$, $f(t_2, t, t_3)$, $f(t, t_3, t_4)$. Note that this B-spline has two segments, although they meet C^∞.

Now suppose we insert the same knot again. Again, we know the knot vector, and again we use blossoming to find the new control points for the B-spline with the knot vector $\{t_0, t_1, t_2, t, t, t_3, t_4, t_5\}$:



We can continue adding this knot until it has multiplicity n. At this point, note that we have the f evaluated at our new knot value:



Thus, we have evaluated the curve by using repeated knot insertion. This is known as the *de Boor algorithm*, and is similar to the de Casteljau algorithm for evaluating Bézier curves. The de Boor algorithm gives us a method for evaluating B-splines without having to convert to Bézier form. Further, one evaluation with the de Boor algorithm costs about the same as one evaluation of a Bézier curve with de Casteljau's algorithm.

Using knot insertion, we can convert the B-spline curve into its Bézier segments by increasing the knots to full multiplicity. Thus, a Bézier curve is actually a special case of a single segment B-spline curve (one where the knots have full multiplicity).

What if we go the other way? Suppose we have a single cubic Bézier curve segment parametrized over $[0, 1]$ and we wish to convert it to B-spline form. Well, we run into a problem: we really want to convert to a knot vector $a < b < 0 < 1 < c < d$ for some values a, b, c, d. However, we can choose *any* such a, b, c, d. Thus, to completely specify the problem, we need to give the additional knots to complete the problem statement. If we have abutting

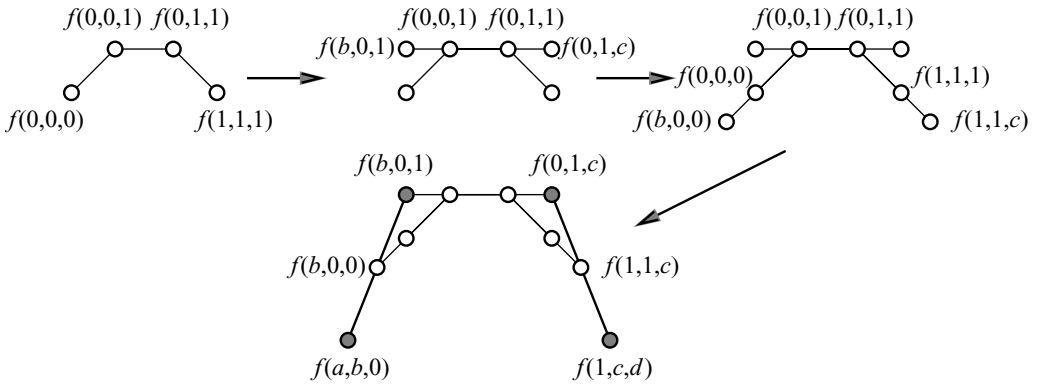


FIGURE 3.3: Converting a Bézier segment to a B-spline segment

segments we wish to compute, then some of these additional knots are already specified (we will have to specify extra ones at the ends and check our continuity level to make sure the conversion process makes sense).

However, if we are given a, b, c, d, then we need to find the B-spline control points $f(a, b, 0)$, $f(b, 0, 1)$, $f(0, 1, c)$, and $f(1, c, d)$ from the Bézier control points $f(0, 0, 0)$, $f(0, 0, 1)$, $f(0, 1, 1)$, and $f(1, 1, 1)$.

The points $f(b, 0, 1)$ and $f(0, 1, c)$ are easy: we just extend the segment $\overline{f(0, 0, 1)f(0, 1, 1)}$ to either side (Fig. 3.3). And once we have $f(b, 0, 1)$, the point $f(a, b, 0)$ is found by extending the segments $\overline{f(0, 0, 1)f(0, 0, 0)}$ to find $f(b, 0, 0)$, and then extending $\overline{f(b, 0, 1)f(b, 0, 0)}$ to find $f(a, b, 0)$. Similarly we can find $f(1, c, d)$.

Figure 3.4 shows two examples of B-spline curves and their control polygons. Both B-splines have uniform knot vectors. Comparing these curves to the Bézier curves appearing in Fig. 2.4, it appears that B-spline curves lie nearer to their control polygon than Bézier curves do (see also Fig. 3.5, left and center). This appearance of closeness is a somewhat illusionary effect of scale: the control polygon for a uniform B-spline is much larger than that of the control polygon for the corresponding Bézier representation; Figure 3.5 (right) shows both the Bézier

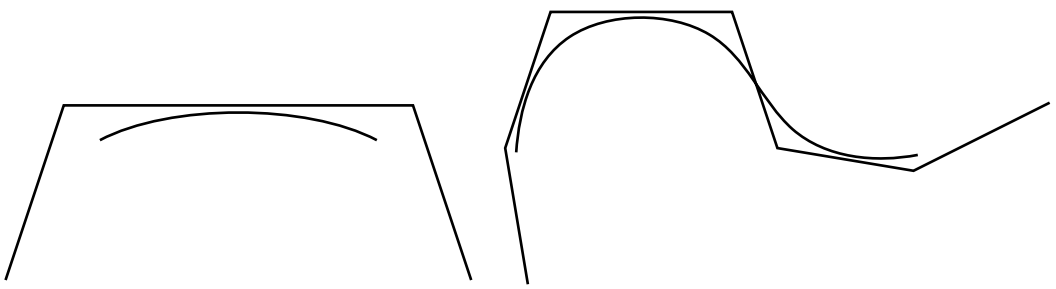


FIGURE 3.4: Some examples of B-spline curves

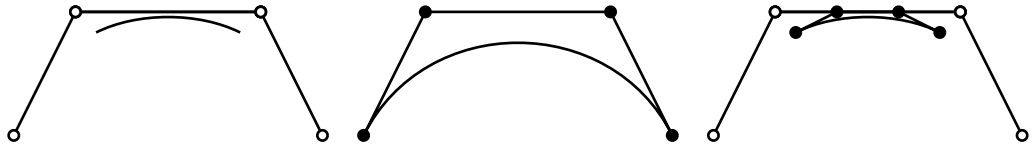


FIGURE 3.5: B-splines (white) and Bézier (black) control points

and B-spline control polygons for the same curve, and here we see that the curve appears closer to the Bézier control polygon.

3.1 IMPLEMENTATIONS

1. Write an interactive 2D cubic B-spline editor with the following functionality:
   - The left mouse button adds a new control point.
   - The middle mouse button is used to move control points.
   - New segments have unit length parameterization (e.g., when adding a new control point, assume the value of any new knot to be one more than the last knot in the knot sequence).
   - There are two display modes:
     - Just the curve.
     - The curve and the control polygon.
   - There should be a reset key/menu option that clears all the control points.

3.2 KNOT MULTIPLICITY

If a knot has multiplicity greater than 1, then some of the B-spline segments are of zero length. This is seen in the definition of the B-spline, since some of our intervals will be of zero length. In general, when we discuss continuity, we will concern ourselves with the nonzero length intervals meeting at a knot value.

What happens if we start with a knot vector having knots of multiplicity more than 1? Alternatively, what happens if we refine a knot to multiplicity greater than 1 and then move the control points?

As an example for a cubic spline, suppose our knot vector is (0, 1, 2, 3, 3, 4, 5, 6). Our control points are $f(0, 1, 2)$, $f(1, 2, 3)$, $f(2, 3, 3)$, $f(3, 3, 4)$, $f(3, 4, 5)$, and $f(4, 5, 6)$. Suppose we extend the segments $\overline{f(1, 2, 3)f(2, 3, 3)}$ and $\overline{f(3, 3, 4)f(3, 4, 5)}$ until they intersect (Fig. 3.6). What blossom value should be assigned to this intersection point?

It might turn out that both calculations yield a blossom value of $f(2, 3, 4)$. If this is the case, we could add this to our control points, delete the extra knot at 3, and remove the control points $f(2, 3, 3)$ and $f(3, 3, 4)$.

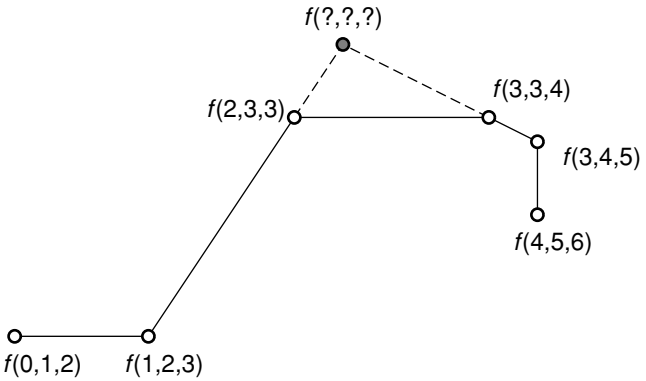
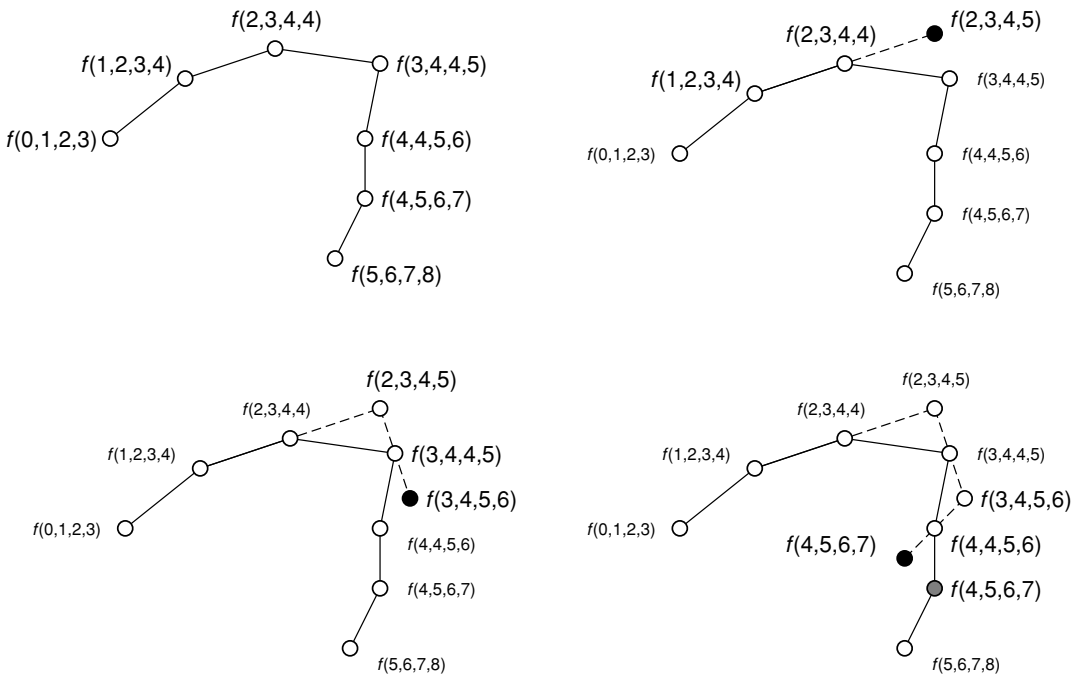


FIGURE 3.6: Cubic B-spline with knot value 3 having multiplicity 2

But suppose we get two different blossom values for this point? First, how is this possible? And second what does it mean?

This is possible because we are really extending two different blossoms (consider the point $f(3, 3, 3)$ to see where the break is). And if you think back to the definition of continuity, this difference means that the B-spline is not C^2 at $t = 3$.

More generally, we can show that if a knot t_i has multiplicity μ, then a degree n B-spline is $C^{n-\mu}$ at t_i. The proof idea is illustrated in the following diagram:



Initially, we have a quartic B-spline with knot vector $(0, 1, 2, 3, 4, 4, 5, 6, 7, 8)$. This B-spline has two segments of nonzero length. One is defined by the control points

$$f(0, 1, 2, 3), \ldots, f(4, 4, 5, 6)$$

the other by

$$f(2, 3, 4, 4), \ldots, f(5, 6, 7, 8)$$

These two segments meet with some level of continuity at $t = 4$. Starting with the former curve segment, let us use its defining control points to determine the location of $f(4, 5, 6, 7)$.

First, we extend the segment $\overline{f(1, 2, 3, 4)f(2, 3, 4, 4)}$ to find the location of $f(2, 3, 4, 5)$. Next, we extend the segment $\overline{f(2, 3, 4, 5)f(3, 4, 4, 5)}$ to find the location of $f(3, 4, 5, 6)$. Finally, we extend the segment $\overline{f(3, 4, 5, 6)f(4, 4, 5, 6)}$ to find $f(4, 5, 6, 7)$.

Thus, the location of $f(4, 5, 6, 7)$ can be calculated from the control points of the first segment. Once these control points are fixed, the location of $f(4, 5, 6, 7)$ from these control points is determined.

But in a B-spline, we are free to place the gray point $f(4, 5, 6, 7)$ in the diagram anywhere we please. And, as seen in the diagram, we can place it at a location different from the one we would calculate the control points of the first segment.

If we were to calculate the Bézier control points of the two segments, we would see that they meet with C^2 continuity at $t = 4$. But since the two segments yield a different location for $f(4, 5, 6, 7)$, the two segments do not meet C^3 at $t = 4$.

Note that we need two segments to talk about continuity. This need for two segments motivates the conditions on which knots have multiplicity greater than one in the following theorem.

Theorem 3.1. *Given a degree n B-spline F with knot vector $t_0 \leq \cdots \leq t_{i-1} < t_i = \cdots = t_{i+k-1} < t_{i+k} \leq \cdots \leq t_{L+2n-2}$, where $0 < k \leq n$, $i \geq n$, and $i + k < L + n - 3$, and where no knot occurs with multiplicity greater than n. Then F is C^{n-k} at t_i, but in general is not C^{n-k+1} at t_i.*

Proof. First, we will prove F is C^{n-k}. Then, we will show that it is not C^{n-k+1}.

We can show F to be C^{n-k} at t_i by showing that the two segments abutting at t_i agree when k of their arguments are t_i, i.e., for all u_i, the following must hold:

$$f\left(t_i^{\langle k \rangle}, u_1, \ldots, u_{n-k}\right) = g\left(t_i^{\langle k \rangle}, u_1, \ldots, u_{n-k}\right)$$

This is seen by noting that the segment defined over $[t_{i-1}, t_i]$ is defined by the control points

$$f(t_{i-n}, \ldots, t_{i-1}), \ldots, f(t_i, \ldots, t_{i+n-1}) = f\left(t_i^{\langle k \rangle}, t_{i+k}, t_{i+k+1}, \ldots, t_{i+n-1}\right)$$

while the segment defined over $[t_i = t_{i+k-1}, t_{i+k}]$ *is defined by the control points*

$$f\left(t_{i-n+k}, \ldots, t_{i-1}, t_i^{\langle k \rangle}\right), \ldots, f(t_{i+k}, \ldots, t_{i+k+n-1})$$

(we consider the interval $[t_i, t_{i+k}]$ *because it is the next nonzero length interval after* $[t_{i-1}, t_i]$*).*

Both segments have the following $n - k + 1$ *control points in common:*

$$f\left(t_{i-n+k}, \ldots, t_{i-1}, t_i^{\langle k \rangle}\right), \ldots, f\left(t_i^{\langle k \rangle}, t_{i+k}, \ldots, t_{i+n-1}\right)$$

Since the knot values

$$t_{i-n+k}, \ldots, t_{i-1}, t_{i+k}, \ldots, t_{i+n-1}$$

have multiplicity no greater than $n - k$*, these control points completely define* f *and* g *when* k *of their arguments are* t_i*, and thus,* F *and* G *meet* C^{n-k} *at* t_i*.*

Now we will show that the B-spline is not C^{n-k+1}*. We will work with the segment defined over* $[t_{i-1}, t_i]$ *whose control points are given above. From* $f(t_{i-n+k+1}, \ldots, t_{i-1}, t_i^{\langle k-1 \rangle})$ *and* $f(t_{i-n+k}, \ldots, t_{i-1}, t_i^{\langle k \rangle})$ *we can compute* $f(t_{i-n+k}, \ldots, t_{i-1}, t_i^{\langle k-1 \rangle}, t_{i+k})$*. Likewise, a proof by induction shows that we can compute the points*

$$f\left(t_{i-n+k+1+j}, \ldots, t_{i-1}, t_i^{\langle k-1 \rangle}, t_{i+k}, \ldots, t_{i+k+j}\right)$$

for $j = 1, \ldots, n - k - 1$*.*

Thus, we can compute the control point $f(t_i^{\langle k-1 \rangle}, t_{i+k}, \ldots, t_{i+n})$*. However, as this point is not in the defining control points of the segment defined over* $[t_{i-1}, t_i]$*, in general the B-spline point will differ from the point computed from this segment, and thus, the B-spline is not* C^{n-k+1} *at* t_i*.*

There is one place at which a multiple knot causes something strange to happen. For a cubic, consider the knot vector (0, 1, 2, 2, 3, 4, 5). Since there are seven knots, it would seem like we have a two-segment spline. However, we really only have one segment—the knot 0 can be omitted, along with the control point $f(0, 1, 2)$. Thus it is pointless to have $t_{n-2} < t_{n-1} = t_n$, since it results in a zero-length segment at the start of the curve, and some number of knots and control points could be omitted without changing the curve. (Although generally not useful, there are times when we might want to give knot t_{n-1} multiplicity greater than 2, such as if we wish to join two B-splines together.) A similar problem occurs at the other end of a B-spline.

Caution: Look at the definition of the B-spline curve. It states which control points are used to define the curve over the interval $[t_{i+n-1}, t_{i+n}]$. More strongly, if we use this set of control points, *then all arguments to the blossom must be in this interval* (by "all arguments," I mean all knots inserted via de Boor's algorithm). Remember, this is a *piecewise* polynomial curve. Likewise, there is a notion of a "piecewise" blossom. What we have done to define a B-spline is play a

notational game. We really should subscript our B-spline F (and likewise, its blossom f) with the interval of interest. Instead, we have hidden this index in the definition of a B-spline.

While we can evaluate the blossom from one of these sets of control points on argument bags that are not all contained within the defining interval, the value may be ill-defined: if we evaluate a different set of control points on the same argument bag, we may get a different value. In fact, except for the cases given by the continuity conditions, we will expect to get different results.

As a final note, uniform B-splines are often sufficient for most modeling tasks, and as we will see in Section 3.4, having a uniform knot vector allows us to use a very simple rendering method. However, if you display the control points and manipulate the curve by moving the control points, it can be somewhat unsettling for the curve to not start at the first control point and end at the last control point. If instead, we use a "nearly uniform" knot vector, with equally spaced knots everywhere except the first and last knots, which we raise to full multiplicity, then the resulting curve interpolates its first and last control points, and we retain most of the simplicity of a uniform knot vector. If desired, we could retain the best of both worlds by displaying the B-spline having the near uniform knot vector, but internally storing it with a uniform knot vector, converting back and forth between the two representations as needed (Fig. 3.3).

3.2.1 Exercises

1. Suppose we have a quadratic B-spline with knot vector (0, 1, 2, 3, 3, 4, 5, 6). Draw a picture similar to Fig. 3.6 for this quadratic B-spline; be sure not to place three consecutive control points on a line. What can you conclude about the continuity of the B-spline at $t = 3$? Sketch the curve on the figure.
2. Suppose we have a linear B-spline with knot vector (0, 1, 2, 3, 3, 4, 5, 6). Draw a picture similar to Fig. 3.6 for this linear B-spline; be sure not to place three consecutive control points on a line. What can you conclude about the continuity of the B-spline at $t = 3$?
3. Using points constructed by the de Boor evaluation algorithm, find formulas for the first and second derivatives of a cubic B-spline curve with knot vector $\{t_0, t_1, t_2, t_3, t_4, t_5\}$ at a point of evaluation $t \in [t_2, t_3]$. Be careful to ensure that the magnitude of the derivative is correct.
4. Give the B-spline control points and corresponding knot vector for the cubic Hermite curve described in Exercise 1 of Section 2.6.5. *Note*: It is inadequate to give the Bézier control points and the corresponding B-spline knot vector; you should remove any knots that can be removed without changing the curve.
5. The continuity theorems about B-splines tell us that if a knot u has single multiplicity, then a B-spline of degree n is C^{n-1} at this point. However, this is clearly an analytic

condition, not a geometric one, since if we make $n+1$ consecutive control points coincident, then we have introduced a segment of zero length (i.e., all derivatives are 0 and the curve is of zero length over a nonzero length interval in the domain) and we can introduce a cusp in the curve.

(a) Suppose that we have the knot sequence (0, 1, 2, 3, 4, 5, 6, 7) for a cubic B-spline f, and further suppose that $f(2, 3, 4) = f(3, 4, 5)$. The theorem on continuity tells us that this curve should be C^2 at $F(3)$. Is the curve geometrically C^2 at $F(3)$? If so, support your statement. If not, give a counterexample.

(b) Now suppose that $f(1, 2, 3) = f(2, 3, 4) = f(3, 4, 5)$. Is the curve geometrically C^2 at $F(3)$? If so, support your statement. If not, give a counterexample.

3.2.2 Implementations

1. Extend your interactive B-spline editor of the previous assignment in the following ways:
   - The right mouse button displays the blossom value of the displayed control point closest to the current mouse position.
   - There are three display modes:
     - Just the curve.
     - The curve and the control polygon.
     - The curve, the control polygon, and the control polygons for the corresponding Bézier curves.
   - Somewhere, you should display the knot vector on a line. You should be able to use the middle mouse button to move existing knots of the knot vector, the left mouse button to add a new knot, and the right mouse button to display the value of the closest knot.
   - There should be a toggle for displaying the blossom arguments of all displayed control points (including the Bézier control points, if they are currently being displayed). Note that if all blossom arguments are being displayed, then the right mouse button does nothing.
   - There should be a "reset" key/menu option that clears all knots and control points.

3.3 TRIANGLE DIAGRAMS

As an alternative means of investigating continuity between B-spline segments, we can use triangle diagrams. Suppose we look at the triangle diagrams for two adjacent B-spline segments. What we find is that the segments overlap (Fig. 3.7). Note that in this diagram I have left out the "normalizing" factor. For example, the two arcs feeding into $f(u_1, u_2, u)$ should be divided by $u_3 - u_0$.

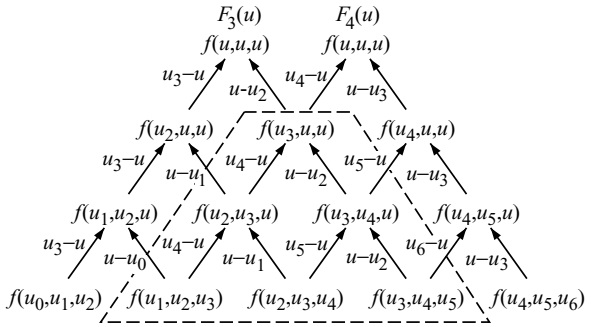


FIGURE 3.7: Overlapping triangle diagrams for adjacent B-spline segments

While we normally think of our B-spline as a single curve, for discussion here it is helpful to give the adjacent segments different names. We will refer to the two segments discussed here as F_3 and F_4, and in Fig. 3.7, we have labeled the apex of their triangle diagrams as such. Although there is a large overlap in their triangles, technically we should not be overlapping their construction, since for a B-spline, F_3 is only defined over $[u_2, u_3]$ while F_4 is only defined over $[u_3, u_4]$.

However, it is valid to overlap the constructions if we evaluate at u_3. Then looking at the arcs into the nodes labeled $f(u, u, u)$, we see that both F_3 and F_4 are completely determined by the overlapping region of the triangles since the top level arcs labeled $u_3 - u$ and $u - u_3$ will have a value of 0 when evaluated at $u = u_3$. If we consider the blossoms of F_3 and F_4, we see that this implies that $f_3(u_3, t_1, t_2) = f_4(u_3, t_1, t_2)$ for all t_1, t_2 and thus, F_3 and F_4 meet with C^2 continuity.

Suppose we have a knot of multiplicity two. Then our triangle diagram looks like Fig. 3.8. Note that there are two nodes labeled $f(u_3, u, u)$. These two nodes cannot be combined to form a node at the next level since they share all three blossom values (to combine them, exactly one argument must differ). However, we see that if we set two arguments of both blossoms to u_3

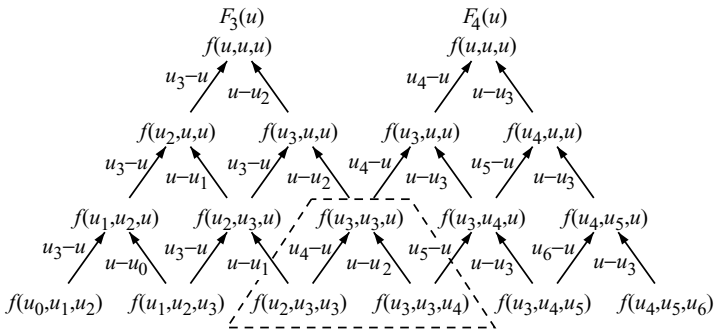


FIGURE 3.8: Overlapping triangle diagrams for adjacent B-spline segments with knot multiplicity 2

then both f_3 and f_4 will match for all values of the final blossom argument (both are completely determined by the region enclosed by dotted lines). Thus, $f_3(u_3, u_3, u) = f_4(u_3, u_3, u)$ for all u, and therefore F_3 and F_4 meet C^1 at u_3. Note however, that they do *not* meet C^2: the construction of $f_3(u_3, u, u)$ depends on the control point $f(u_1, u_2, u_3)$ while the construction of $f_4(u_3, u, u)$ depends on the control point $f(u_3, u_4, u_5)$. Since we are free to set these two control points to arbitrary values, we know in general that $f_3(u_3, u, u) \neq f_4(u_3, u, u)$, and thus, F_3 and F_4 do not meet C^2.

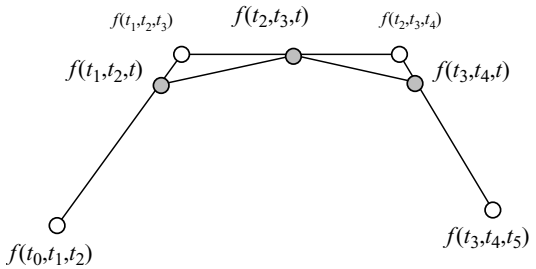
3.3.1 Exercise

1. Draw the overlapping triangle diagram for a cubic B-spline with knot vector $\{a, b, c, d, d, d, e, f, g\}$. What can you conclude about the continuity of the B-spline at $t = d$?

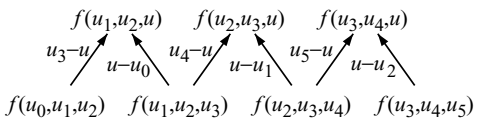
3.4 KNOT INSERTION

We have used knot insertion with B-splines to evaluate a B-spline curve and to find the Bézier control points for a B-spline. Often we find that many operations we look at in another manner can also be described with knot insertion. In general, knot insertion techniques are attractive since they commonly give fast methods for performing various tasks. Knot insertion is a technique that is readily described by triangle diagrams, and typically drawing the triangle diagram is all you need to see the desired result. This section presents a few knot insertion algorithms that are readily explained with triangle diagrams; see [1, 2, 3, 10] for additional details and algorithms.

The first knot insertion technique we will consider is the first step of the de Boor algorithm, where we have a cubic segment with knot vector $u_0, u_1, u_2, u_3, u_4, u_5$ and we insert the knot t between knots u_2 and u_3:

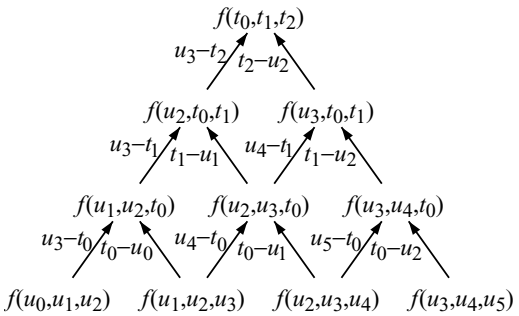


This form of knot insertion is known as Boehm's algorithm. It has a triangle diagram of



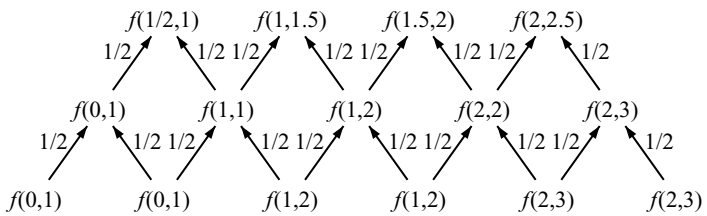
and from the diagram, we immediately see the control points for the knot vector $\{u_0, u_1, u_2, t, u_3, u_4, u_5\}$.

A second knot insertion algorithm is the *Oslo algorithm*, where we insert different knots at each level of the triangle. Starting with the knot vector $\{u_0, u_1, u_2, u_3, -u_4, u_5\}$ and inserting knots t_0, t_1, t_2 with $u_2 < t_0 < t_1 < t_2 < u_3$ we get



The idea of the Oslo algorithm is to refine a B-spline to be over a denser knot vector; the goal is to do the refinement efficiently. In this triangle diagram, the left and right edges give the control points for the B-spline with knot vector $\{u_0, u_1, u_2, t_0, t_1, t_2, u_3, u_4, u_5\}$. Goldman explores more efficient variations of the Oslo algorithm with triangle diagrams [3].

A third knot insertion algorithm that we will (mostly) show using the triangle diagram is the Lane–Riesenfeld algorithm [12, 23]. The Lane–Riesenfeld algorithm starts with a uniform B-spline and inserts knots midway between existing knots. The algorithm is to replicate each control point, and average n times (where n is the degree of the B-spline). For quadratics, the triangle diagram is



At the bottom of the diagram, we see the quadratic control points (with knot vector $\{0, 1, 2, 3\}$) replicated twice, and on the top we see that we have the control points for the same function f but with knot vector $\{0.5,\ 1,\ 1.5,\ 2,\ 2.5\}$ (we lose the knots 0 and 3 when we insert knots outside the interval over which the curve is defined).

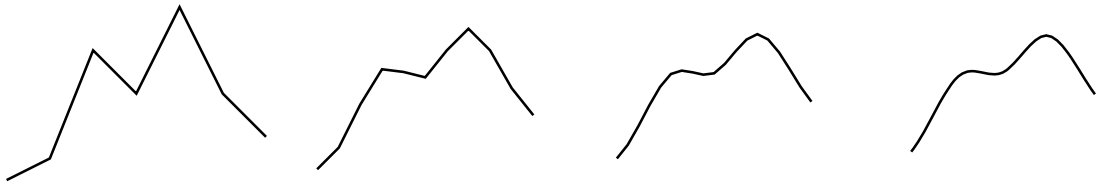


FIGURE 3.9: Three subdivisions using the Lane–Riesenfeld algorithm

Unfortunately, the triangle diagram does not work as well when trying the describe the degree 3 or higher variation of the Lane–Riesenfeld algorithm, although the algorithm itself readily extends to higher degrees, with one averaging step per degree.

As a side note, realize that the Lane–Riesenfeld algorithm is a particularly simple way to draw uniform B-splines: at each step, you simply average points together. As you repeatedly apply the Lane–Riesenfeld algorithm, the refined polygon converges to the curve. While exact convergence only occurs after infinite number of refinements, we can achieve a good approximation after four or five levels of refinement, as illustrated in Fig. 3.9 for a cubic B-spline refined three times.

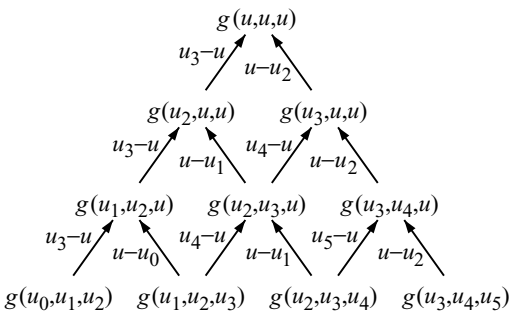
3.4.1 Implementations

1. Implement the Lane–Riesenfeld algorithm for rendering uniform B-splines.

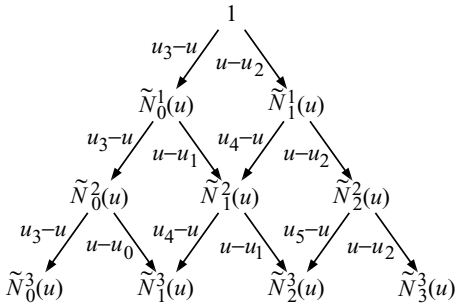
3.5 B-SPLINE BASIS FUNCTIONS

We know the B-spline control points are being blended with some polynomial blending functions. If all knots are of full multiplicity, then we have a piecewise Bézier curve and the blending functions are the Bernstein polynomials. But what if the knots are not of full multiplicity?

For simplicity, assume all knots have multiplicity one. For the de Casteljau algorithm, we constructed a dataflow diagram. Suppose we do the same for the de Boor algorithm for a single segment cubic B-spline with knot vector u_0, u_1, u_2, u_3, u_4, u_5:



If we trace all paths from the root to a control point, we get the contribution of that control point to the final value. Thus, if we retain the edge labels, feed 1 into the root, and run the flow backwards, we get the B-spline basis functions:



(Technically, we have only found blending functions as we have not shown that these functions form a basis. While true, I will not prove that here.) The dataflow diagram leads to the following recurrence:

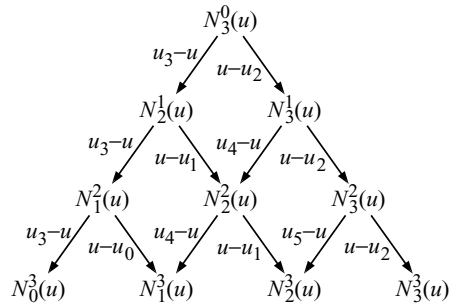
$$\tilde{N}_i^m(u) = \frac{u - u_{n-m+i-1}}{u_{n+i-1} - u_{n-m+i-1}} \tilde{N}_{i-1}^{m-1}(u) + \frac{u_{n+i} - u}{u_{n+i} - u_{n-m+i}} \tilde{N}_i^{m-1}(u) \qquad (3.1)$$

The "1" at the top of the dataflow diagram could also be written as $\tilde{N}_0^0(u)$, with $\tilde{N}_i^0(u) = 1$ if $u_i < u < u_{i+1}$ and 0 otherwise. This is a form of what is known as the Cox–de Boor–Mansfield B-spline recurrence relation, which is more commonly written as

$$N_i^m(u) = \frac{u - u_{i-1}}{u_{i+m-1} - u_{i-1}} N_i^{m-1}(u) + \frac{u_{i+m} - u}{u_{i+m} - u_i} N_{i+1}^{m-1}(u) \qquad (3.2)$$

with $N_i^0(u)$ defined to be 1 if $u_{i-1} \le u \le u_i$ and 0 otherwise. This latter form is preferred, as we have removed the dependency on n.

N and $\tilde{N}$ in the above two recurrence formulas would refer to the same functions, except for different indexes and the dependency of $\tilde{N}$ on n. We can also construct a triangle for the second recurrence:



In the previous version, the subscript was constant when moving up and to the right. Now the subscript is constant when moving up and to the left. While the former dataflow diagram has somewhat nicer subscripts in the diagram, the former recurrence's [Eq. (3.1)] dependency on n makes the latter form [Eq. (3.2)] preferable.

With these basis functions, we can now write the following expression for our B-spline curve:

$$F(t) = \sum_{i=0}^{L+n-1} p_i N_i^n(t) \tag{3.3}$$

where N_i^n are the B-spline basis functions of Eq. (3.2) with the knot vector and control points p_i of Definition 3.1.

There are several things a bit unusual about these dataflow diagrams and the recurrences. For example, looking more closely at $N_0^3(u)$ as expanded by Eq. 3.2, we see that it is given by

$$N_0^3(u) = \frac{u - u_{-1}}{u_{3-1} - u_{-1}} N_0^2(u) + \frac{u_3 - u}{u_3 - u_0} N_1^2(u)$$

From this expression, we see two odd things: first, N_0^3 depends on knot u_{-1} and second, N_0^3 depends on N_0^2. However, the node for N_0^2 (and the arc to it that would be labeled $u - u_{-1}$) are missing from the dataflow diagram. Similarly, at all the other nodes along the edge of the dataflow diagram, the recurrence indicates that there is a node missing from the dataflow diagram (and at N_3^3, an additional unspecified knot u_6).

The reason for these omissions from the dataflow diagram is that we are only interested in this dataflow diagram for $u_2 < u < u_3$. With u in this range, the missing nodes all have value 0 and do not contribute to the values of the nodes in the diagram. Since the nodes have 0 value, we omit them from the diagram.

The additional knots that seem to appear on either side (u_{-1} and u_6 in our cubic example) may also be omitted. However, if you write code to compute the blending functions, it simplifies the code to have those knots available (basically by moving two conditionals out of the code to implement Eq. 3.2 and into the beginning of a function for $N_i^n(t)$). Thus, in many implementations of B-splines (such as OpenGL), you have to specify an additional two knots in the knot vector. These *phantom knots* are given zero weight by the zero basis functions, and you are pretty much free to set them as you like, although commonly you would set them to the value of the first and last knots or have them extend a sequence of the other knots. For example, for a single cubic B-spline segment with uniform knots, where we have been using 0, 1, 2, 3, 4, 5 as our knot vector, with phantom knots you would use 0, 0, 1, 2, 3, 4, 5, 5 or −1, 0, 1, 2, 3, 4, 5, 6.

Another bothersome fact about both recurrences is that N^m is not an affine combination of two of the N^{m-1}s. This is unimportant, however: we already know that $\sum N_i^m(u) = 1$ by the de Boor evaluation algorithm.

So far we have looked at the basis functions for the points used by the de Boor construction over one interval. Since the other control points are not used in this construction, we set their basis functions to zero over this interval.

We can turn this question around, however: what does the basis function for a particular control point look like over the entire real line? We already know that we can restrict our question to a subset of the real line, since the basis function will be zero outside of this set. In particular, for cubics, the basis function for the control point $f(t_i, t_{i+1}, t_{i+2})$ is used in the de Boor construction for $t \in [t_{i-1}, t_{i+3}]$. Thus, the basis function is nonzero over four intervals. Further, on any particular interval, only four basis functions have nonzero weight.

For simplicity, we will consider the case when the knots are equally spaced (this is known as a *uniform B-spline*). Look at the de Boor construction for t and $t + 1$. In terms of the blending functions, the same four functions are being constructed. They are, however, being used to weight different control points. As we move from t_{i-1} to t_i, one blending function is used to weight $f(t_i, t_{i+1}, t_{i+2})$. As we move from t_i to t_{i+1}, a second blending function is used. And from t_{i+1} to t_{i+2}, a third is used. Finally, from t_{i+2} to t_{i+3}, the fourth blending function weights $f(t_i, t_{i+1}, t_{i+2})$.

Since the blending functions are used to weight $f(t_i, t_{i+1}, t_{i+2})$, a natural question to ask is "What continuity do the blending functions have?" We know a few facts about them and can guess a few more (but I will not prove them):

- They are nonnegative (this is seen from the recursive definition of the blending functions).
- They sum to 1 (this is a result of the de Boor construction).
- Neighboring blending functions meet C^2 since the curve is C^2.
- The first blending function starts at zero and furthermore, meets the zero line C^2 (since the basis function for our point is zero outside of the interval $[t_{i-1}, t_{i+3}]$).

The above facts can all be proven, and hold for an arbitrary knot vector where all knots have multiplicity one.

The first two of the above facts show that a B-spline curve possesses the convex hull property. In fact, it possesses something stronger: each curve segment lies in the convex hull of the control points defining it. This is useful: if we create a long string of control points, then the curve will have roughly the shape of the control polygon.

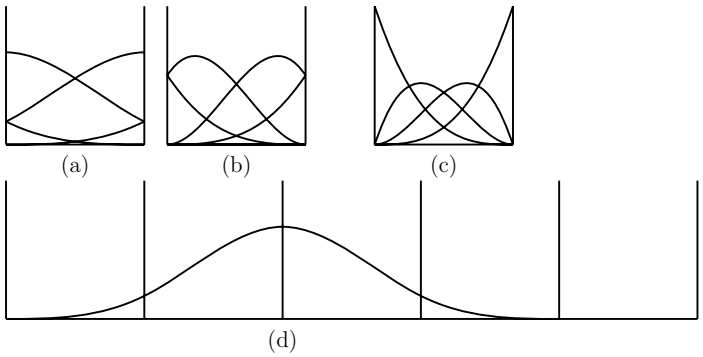


FIGURE 3.10: Some B-spline basis functions

Figure 3.10 shows some examples of the B-spline basis functions. In Fig. 3.10(a), we see the four basis functions used to weight the control points of a cubic B-spline curve with a uniform knot vector {0, 1, 2, 3, 4, 5}; in Fig. 3.10(d), we see the a single blending function used to weight a single control point for a cubic B-spline with uniform knots. The vertical lines indicate where the knots are located; note that each subcurve in (d) is a translation of one of the basis functions seen in (a). As suggested above, the curve appearing in Fig. 3.10(d) is a piecewise cubic curve with the pieces meeting with C^2 continuity at the knots.

In Fig. 3.10(b), we see the cubic B-spline basis functions for the knot vector {0, 1, 1, 2, 2, 3}. Notice while in Fig. 3.10(a), only one basis function is zero at the ends of the interval, when a knot has multiplicity 2, then two of the basis functions are zero at duplicated knot. In Fig. 3.10(c), we see the cubic B-spline basis functions for the knot vector {0, 0, 0, 1, 1, 1}, which turn out to be the cubic Bernstein polynomials. And in this case, we see that three of the basis functions are zero at the knots, in accord with their having multiplicity 3.

3.5.1 Exercises

1. Prove that Eqs. (3.1) and (3.2) are equivalent.
2. Under what conditions will a B-spline basis function obtain the value of 1?

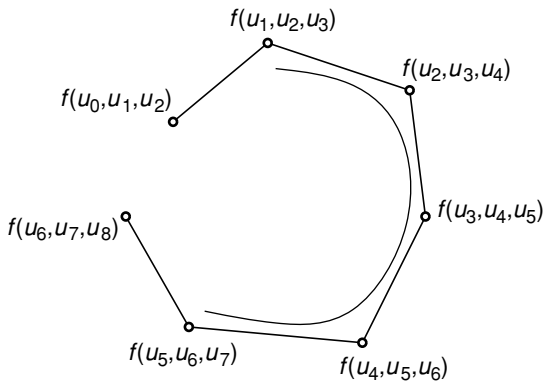
3.5.2 Implementations

1. Write a program to plot the B-spline basis functions for the following degree curves and knot vectors:
   (a) Degree 3, knot vector (0, 1, 2, 4, 8, 16).
   (b) Degree 3, knot vector (0, 1, 2, 3, 4, 4, 5, 5, 5, 6, 7, 8).
   (c) Degree 4, knot vector (0, 2, 3, 3, 3, 4, 5, 7, 10).

3.6 CLOSED B-SPLINES

Suppose we want to construct a closed curve with a B-spline. If we only want C^0 continuity at the "end," we merely need to raise the first and the last knot to full multiplicity and set the last control point to the first control point. By "first knot," I mean the knot corresponding to the start of the first interval of the B-spline. Similarly for the "last knot."

However, suppose we want our B-spline to meet with full continuity at the start/end of the curve. Suppose, for example, we have the cubic B-spline shown below, over the knot vector $(u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8)$ and we wish to close the curve by adding additional knots and control points. Where do we place these new knots and control points so that the curve is C^2 everywhere?



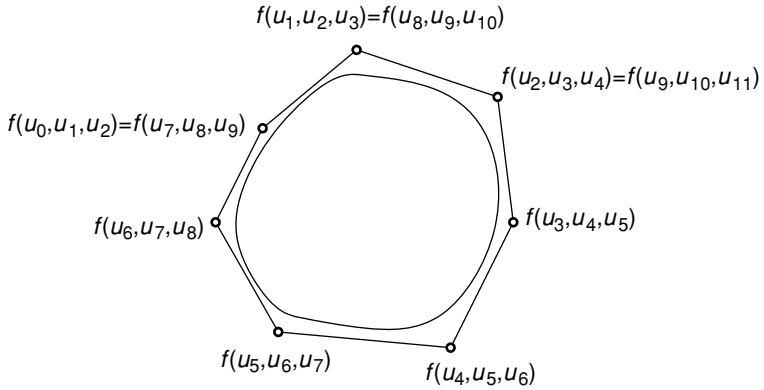
One way to think of the problem is that we wish to add a segment to the beginning of the curve. Then in this case, we add a new knot, u_{-1}, and a new control point, $f(u_{-1}, u_0, u_1)$, and then the new curve segment defined by $f(u_{-1}, u_0, u_1)$, $f(u_0, u_1, u_2)$, $f(u_1, u_2, u_3)$, and $f(u_2, u_3, u_4)$ over the interval $[u_1, u_2]$ will meet our B-spline with C^2 continuity. For simplicity, call this segment F_1.

Thus, if we wish to close our curve so as to meet C^2, we should place the final control points at these locations and "use the same knots." Of course, we cannot actually use the same knots since our knot sequence must be nondecreasing, but for the new segment to trace the same curve as F_1, we must have the same knot spacing.

Thus, our new knot vector will be $(u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, u_{10}, u_{11})$ with the following constraints:

$$\begin{aligned} u_{10} - u_{11} &= u_3 - u_4 \\ u_9 - u_{10} &= u_2 - u_3 \\ u_8 - u_9 &= u_1 - u_2 \\ u_7 - u_8 &= u_0 - u_1 \end{aligned}$$

and the last three control points must be identical to the first three control points as illustrated below:



However, our solution is not quite fair in two ways: first, we can rewrite our constraints on the knots as

$$\begin{aligned} u_8 &= u_7 - u_0 + u_1 \\ u_9 &= u_8 - u_1 + u_2 \\ u_{10} &= u_9 - u_2 + u_3 \\ u_{11} &= u_{10} - u_3 + u_4 \end{aligned}$$

Thus, u_8 is determined once we set u_7. But in the problem description above, we are given u_8. We can address this problem by adding an extra knot and control point.

The second way in which our solution is unfair is that we are adding multiple segments to get the ends to meet with C^2 continuity (rather than just adding a single segment). In the construction above, we actually add three segments. If we want no constraints on u_8, we would have to add four segments. Thus, in general we will have to add several segments to close the curve, or we will have to construct the last control points and segments with strong constraints to achieve the continuity we desire.

It should be clear how to generalize the above construction/set of constraints to higher order B-splines. Further, if you wish to have lower order continuity, then you merely need to duplicate the "first" knot (u_2 in the above example) and let the construction proceed in the same fashion as above.

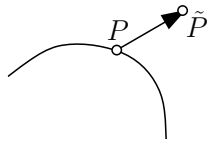
3.7 MODELING WITH POLYNOMIAL AND SPLINE CURVES: DIRECT MANIPULATION

From a mathematical viewpoint, splines and their control points are great. From a programming viewpoint, you will want to view and manipulate control points to ensure that your software

is working. From a user interface viewpoint, control points are terrible. A designer commonly does not have a strong mathematics background, and does not want to learn about splines or their control points. Further, a designer does not like manipulating control points as an indirect handle on a curve.

However, there is no need to show control points in a final software package (although they have been around long enough that you might want to have the option of "turning them on" for those who are used to them). Instead, one can use methods to construct a spline that interpolates or approximates a set of points [7]. This still leaves the issue of how to modify a curve once it has been created.

A simple way to edit a Bézier or B-spline curve is to move the control points of the curve. If we do not display the control points, we would want to select a point P on the curve, move P to $\tilde{P}$, and to adjust the control points of the curve so that the new curve passes through $\tilde{P}$. If $F(t)$ is our original curve, $\tilde{F}(t)$ is the modified curve, and if $P = F(\bar{t})$, then our goal is to find $\tilde{F}$ such that $\tilde{P} = \tilde{F}(\bar{t})$.



The problem as stated is under constrained, and a trivial (but not very useful) solution exists: if $\vec{v} = \tilde{P} - P$, then (for $F(t) = \sum P_i N_i(t)$) set $\tilde{P}_i = P_i + \vec{v}$, and the curve $\tilde{F}(t) = \sum \tilde{P}_i N_i(t)$ will have the property that $\tilde{F}(\bar{t}) = \tilde{P}$. This results in a translation of the entire curve by $\vec{v}$, which is not what we were looking for.

Abstractly, we want to move the point $P = F(\bar{t})$ to $\tilde{P}$ while minimizing the change in $F(t)$. What the minimization criteria should be is less clear. However, if we set

$$\tilde{P}_i = P_i + \vec{v}\frac{N_i(\bar{t})}{\sum_j N_j(\bar{t})^2}$$

then $\tilde{F}(t)$ will have the desired interpolation property:

$$\begin{aligned}
\tilde{F}(\bar{t}) &= \sum_i \tilde{P}_i N_i(\bar{t}) \\
&= \sum_i \left(P_i + \vec{v}\frac{N_i(\bar{t})}{\sum_j N_j(\bar{t})^2} \right) Ni(\bar{t}) \\
&= \sum_i P_i N_i(\bar{t}) + \sum_i \vec{v}\frac{N_i(\bar{t})}{\sum_j N_j(\bar{t})^2} N_i(\bar{t}) \\
&= F(\bar{t}) + \vec{v}
\end{aligned}$$

Further, this approach minimizes the change of the control points of F [4]. Additionally, you can find methods for modifying the tangent to the curves [8].

3.7.1 Implementations

1. Modify your B-spline editor to allow for direct manipulation of the B-spline curve.

3.8 NURBS

In this lecture, we are studying polynomial curves and surfaces. As such, the spline curves we have investigated are *NUBS*—nonuniform B-splines, with the "nonuniform" referring to the knot vector. Most modeling applications allow you to use *NURBS*—nonuniform, rational B-splines—a generalization of NUBS. The "rational" in NURBS refers to our curve being made up of pieces that are ratios of polynomials instead of being a single polynomial.

Mathematically, a NURBS curve has the form

$$F(t) = \frac{\sum_{i=0}^{L+n-1} w_i p_i N_i^n(t)}{\sum_{i=0}^{L+n-1} w_i N_i^n(t)} \tag{3.4}$$

where w_i's are *weights*, with $w_i \in \mathcal{R}$. If all the weights are equal, then this expression for a NURBS curve simplifies to that of the NUBS curve described in Eq. (3.3) (i.e., the curve is a polynomial curve). Intuitively, the weights are additional shape parameters: increasing weight w_i pulls the curve closer to control point p_i and decreasing w_i pushes the curve further from control point p_i. Beyond that, it is unclear how to set the weights without careful mathematical study. And while there are some clever tricks you can play by setting some of the weights as negative, one has to be careful to avoid a set of weights that causes the numerator of Eq. (3.4) to become 0 over the interval $[t_{n-1}, t_{L-1}]$.

The motivation for using rational splines came from drafting, where conic sections were commonly used. Parametric polynomial curves can only represent parabolas, while rationals can represent all conic sections. However, conic sections can be approximated to high tolerance with a NUBS curve, and you have plenty of degrees of freedom in a NUBS curve, making the extra degrees of freedom in the weights of a NURBS curve superfluous. Unless you have a specific need for the weights of a NURBS curve, you are probably best off setting all the weights to 1 and modeling with a NUBS curve. Indeed, even the knot vector of a NUBS is likely more than you need for most modeling applications. Except for setting the end knots to full multiplicity to have an end point interpolating B-spline, you may be best off using a uniform B-spline curve.

CHAPTER 4

Surfaces

In addition to modeling curves, we would like to model surfaces. While we can model spline surfaces, the results are not as satisfying as they were for curves. Part of the problem is that we have more issues to deal with when modeling surfaces: what shape surfaces patches do we need and can we easily model? How do we join two patches together smoothly? How do we create a network of smoothly joined patches?

These plus other issues make surface modeling a complex topic. In this chapter, we will only investigate some of the basics that are readily analyzed for splines. In particular, we will look at triangular and tensor-product spline surfaces, and look at conditions for joining patches with parametric continuity.

Missing from the discussion here is geometric continuity, creation of a network of patches that meet smoothly, and issues of surface quality. The material presented in this chapter should give you enough background to study other sources that address those issues.

4.1 TRIANGULAR SURFACE PATCHES

We begin by looking at barycentric coordinates, which are a form of coordinate system relative to a simplex. A *simplex* is set of $d+1$ points that are linearly independent. For example, two points that form a line segment, three points that form a triangle, or four points that form a tetrahedron. In each of these cases, for the points to form a simplex, their positions must not be degenerate (e.g., the two points of the line segment must be unique, the three points of the triangle must not be colinear, the four points of tetrahedron must not be coplanar). Our focus will be on triangular simplices.

We can express any point P that lies on the space (or subspace) spanned by the simplex with coordinates relative to that simplex. When we weight the elements of the simplex with these coordinates, we recover the point P. Further, the sum of the weights of these coordinates is 1. These are *barycentric coordinates*.

For example, consider a triangle $\triangle ABC$ and an arbitrary point P in the plane of the triangle. We can find scalar values a, b, c such that $P = aA + bB + cC$, where $a + b + c = 1$. The formulas for a, b, c turn out to be simple: to compute a, we take the ratio of the area of $\triangle PBC$ to the area of triangle $\triangle ABC$ (see Fig. 4.1).

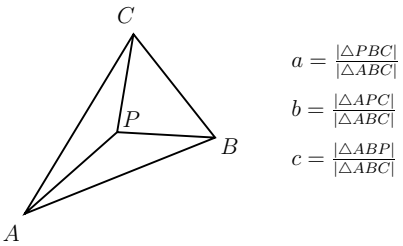


FIGURE 4.1: Barycentric coordinates of P relative to $\triangle ABC$

If we use the signed area of the triangles, giving positive sign to the area if the vertices are in counter-clockwise orientation and negative sign if they have clockwise orientation, then the formulas for barycentric coordinates work for any point in the plane of the triangle (although we should insist that our initial triangle have counter-clockwise orientation). If P lies on the inside of $\triangle ABC$, then all of its barycentric coordinates will be positive; if P lies on the outside of $\triangle ABC$, then some of its barycentric coordinates will be positive and some will be negative.

We will use barycentric coordinates to generalize the Bernstein polynomials. We originally defined the univariate Bernstein polynomials as

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i$$

Here t is an element of the real numbers. Note that this definition is symmetric with respect to t and $1-t$. With this definition of the Bernstein polynomials, the corresponding Bézier curves were most naturally defined over the interval $[0, 1]$. When we wanted to extend the definition to the interval $[a, b]$, we had to modify the Bernstein polynomials.

Barycentric coordinates give us an alternative approach. Rather than let t vary over absolute positions over the real line, we instead look at the interval of interest and write each point along the real line as an affine combination of the endpoints of that interval. It turns out that the affine weights are the desired barycentric coordinates. Thus, if our interval is $[a, b]$, we write a point t as $t = u_0 \cdot a + u_1 \cdot b$, where $u_0 + u_1 = 1$. If our interval is $[0, 1]$, note that $u_0 = 1 - t$ and $u_1 = t$.

With respect to barycentric coordinates, our Bernstein polynomials become

$$B_i^n(u_0, u_1) = \binom{n}{i}u_0^{n-i}u_1^i$$

Next note that $\binom{n}{i} = n!/(i!(n-i)!) = \binom{n}{n-i}$. This definition is symmetric in i and $n-i$. We can make the definition fully symmetric as follows:

$$\binom{n}{\vec{i}} = \frac{n!}{i_0!i_1!}$$

where $\vec{i} = (i_0, i_1)$ and $i_0 + i_1 = n$. With this new definition of "choose," the Bernstein polynomials become

$$B^n_{\vec{i}}(u_0, u_1) = \binom{n}{\vec{i}} u_0^{i_0} u_1^{i_1}$$

This definition is fully symmetric (note that we made the arbitrary choice that $i_0 = n - i$ and $i_1 = i$ between our two definitions). To simplify the notation a bit more, we will usually write $B^n_i(u)$ and assume the expansion of u into barycentric coordinates.

At this point, nothing has actually changed (except for simplifying the Bernstein polynomials over arbitrary intervals). The curves are still the same, etc. (although see below for some issues with the derivative). However, we can write a point p in d-space in barycentric coordinates with respect to some d-simplex in that space. For example, in two space, the two simplex is a triangle $\triangle p_0 p_1 p_2$. Any point p in this two space can be written in barycentric coordinates with respect to this triangle:

$$p = u_0 p_0 + u_1 p_1 + u_2 p_2$$

where $u_0 + u_1 + u_2 = 1$.

What we want is a generalized definition of the Bernstein polynomials to d-dimensional spaces. The following definition works:

$$B^n_{\vec{i}}(u) = \binom{n}{\vec{i}} u_0^{i_0} \cdot \cdots \cdot u_d^{i_d}$$

where $u = (u_0, \ldots, u_d)$ are the barycentric coordinates of a point in our space with respect to the given simplex, and where $\vec{i} = (i_0, \ldots, i_d)$ with $\sum_{j=0}^{d} i_j = n$ and $\binom{n}{\vec{i}} = n!/(i_0! \cdot \cdots \cdot i_d!)$.

These generalized Bernstein polynomials have the properties we want:

- They sum to 1.
- They are nonnegative if our point is in the simplex.
- They have a recursive definition

$$B^n_{\vec{i}}(u) = \sum_{j=0}^{d} u_j B^{n-1}_{\vec{i}-\vec{e}_j}(u)$$

  where $\vec{e}_j$ is the multiindex with zero in every component except the jth component which is 1.
- They form a basis for the d-variate polynomials.

The second property is easy to see; we will not prove the other three, though.

Note also that if $u = (1, 0, \ldots, 0)$ then $B^n_{(n,0,\ldots,0)}(u) = 1$ and the remaining $B^n_{\vec{i}}(u) = 0$. We get a similar result if $u_j = 1$ for $j = 1, \ldots, d$.

Now we define our function as

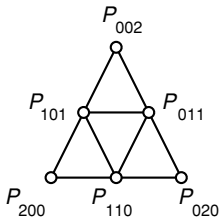
$$B(u) = \sum_{\vec{i},\ |\vec{i}|=n} P_{\vec{i}} B^n_{\vec{i}}(u)$$

This will be a mapping from a d-simplex to a piece of a simplicial d-fold patch.

When $d = 1$ (the curve case), our control points are now doubly indexed: $P_{n,0}$, $P_{n-1,1}, \ldots, P_{0,n}$. Otherwise our curves are the same, with one cautionary note: we need to be careful about derivatives. The barycentric form hides the size of the domain. But the domain size plays a part in the derivatives. Thus, while we can think of these "barycentric curves" as being "affine curves," we really need them to be "Euclidean curves" to treat the derivative as we have been.

Let us look at some examples when $d = 2$ and our domain is a triangle. If $n = 1$, then our control points will be P_{100}, P_{010}, P_{001}, and our blending functions will be u_0, u_1, and u_2. Thus, our image will be a triangle. When $u_0 = 1$, we know that $u_1 = u_2 = 0$ and $B(u) = P_{100}$. Similarly for when $u_1 = 1$ and $u_2 = 1$.

If $n = 2$, then our control points will be P_{002}, P_{011}, P_{020}, P_{101}, P_{110}, and P_{200}. We can visualize it as shown in the following diagram:



In this diagram, I have labeled the control points with the $\vec{i}$ subscript; although the diagram is 2D, control points are in 3-space. We know that the surface will interpolate the corner control points (i.e., P_{200}, P_{020}, and P_{002}). And as we allow u to vary over the domain triangle, the image will trace out a triangular surface patch.

Suppose that we evaluate along one edge of the domain triangle. For example, let $u_2 = 0$. Then our basis functions will be zero if they have a term of u_2 raised to any power greater than 0. Thus, only the basis functions weighting P_{200}, P_{110}, and P_{020} will be nonzero. Since u_2 is raised to the zero power, we can think of these as being functions of only u_0 and u_1. All of this means that as we vary μ along the edge of our domain triangle, we will trace out a Bézier curve in space.

Note further that if we know the directional derivative in two nonparallel directions at a point on the surface, then we have completely characterized the first derivative at that point on the surface. In particular, at any corner control point, we know the directional derivative in two directions (one for each boundary curve that abuts this control point). Thus, we have a

complete characterization of the first derivative at the corner points, and further, we see that the tangent plane at a corner is given by the panel abutting the corner. For example, the tangent plane at P_{002} is given by the plane spanned by P_{002}, P_{011}, and P_{101}.

To further analyze triangular Bézier patches, we will extend the blossoming theorem to higher dimensions.

4.1.1 Blossoming

Theorem 4.1. *The blossoming principle*

Let $F : \mathcal{P} \mapsto \mathcal{Q}$ be a degree n polynomial, where $\mathcal{P}$ and $\mathcal{Q}$ are affine spaces.

Then there exists a unique map $f : \mathcal{P}^n \mapsto \mathcal{Q}$ such that

1. *f is symmetric,*
2. *f is multiaffine,*
3. *$f(u, \ldots, u) = F(u)$.*

f is said to the the polar form *or multiaffine blossom of F.*

Proof. Exercise 1 of Section 4.1.2. (The proof is straightforward albeit notationally involved if we express F in the barycentric Bernstein basis.)

The difference between this and the earlier blossoming principle is that before we assumed that the dimension of $\mathcal{P}$ was 1. Now we will let it be arbitrary. Thus, the arguments to F (and of f) have dimension greater than 1.

But again, given a domain simplex (e.g., a triangle $\triangle ABC$) we completely specify the blossom if we know its values for all argument combinations taken from the vertices of the simplex (e.g., $\{A, B, C\}$). Thus, in the case of degree 2 polynomials, we know f if we know the values $f(A, A)$, $f(A, B)$, $f(A, C)$, $f(B, B)$, $f(B, C)$, $f(C, C)$.

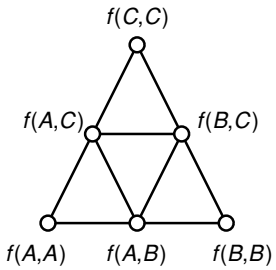
For example, for a 2-simplex domain of $\triangle ABC$, suppose we want to evaluate $F(u) = f(u, u, \ldots, u)$ where F is a degree n polynomial, and u has barycentric coordinates (u_a, u_b, u_c) relative to our domain simplex. Then

$$\begin{aligned} f\left(u^{\langle n\rangle}\right) &= f\left(u_a A + u_b B + u_c C, u^{\langle n-1\rangle}\right) \\ &= u_a f\left(A, u^{\langle n-1\rangle}\right) + u_b f\left(B, u^{\langle n-1\rangle}\right) + u_c f\left(C, u^{\langle n-1\rangle}\right) \\ &\vdots \\ &= \sum_{\vec{i},\ |\vec{i}|=n}^{n} \binom{n}{\vec{i}} u_a^{i_0} u_b^{i_1} u_c^{i_2} f\left(A^{\langle i_0\rangle}, B^{\langle i_1\rangle}, C^{\langle i_2\rangle}\right) \\ &= \sum_{\vec{i},\ |\vec{i}|=n}^{n} f\left(A^{\langle i_0\rangle}, B^{\langle i_1\rangle}, C^{\langle i_2\rangle}\right) B_{\vec{i}}^{n}(u). \end{aligned}$$

Since the generalized Bernstein polynomials form a basis, this further tells us that

$$P_{ijk} = f(A^{\langle i \rangle}, B^{\langle j \rangle}, C^{\langle k \rangle})$$

Using the blossom labels for the control points, we get the following diagram for a quadratic surface in triangular Bézier patch form:

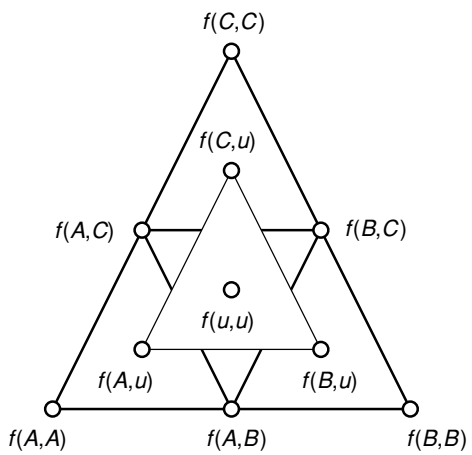


This also indicates how we can evaluate our surface patch. Looking at the diagram, we see that any "upward" pointing triangle has vertices whose blossom values differ in only one argument. The argument that differs takes the values A, B, C over the three vertices. If we weight the control point having D_i as the differing argument with weight u_i, then we can bring the sum to the inside of the blossom, and this sum equals u:

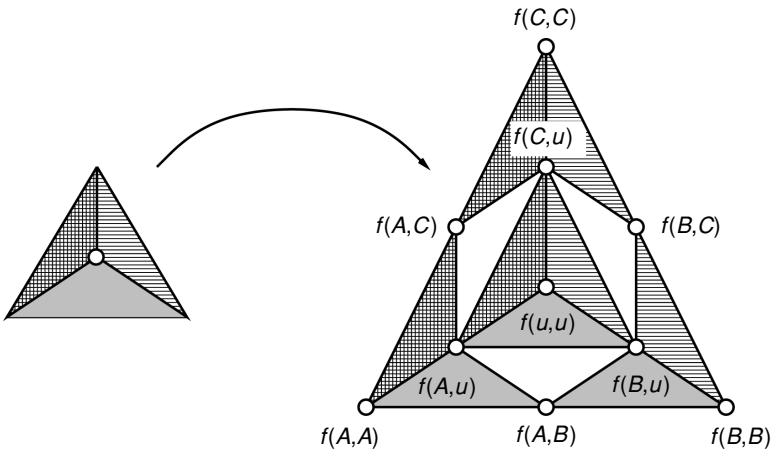
$$\begin{aligned} u_a f(A, A) + u_b f(A, B) + u_c f(A, C) &= f(A, u_a A + u_b B + u_c C) \\ &= f(A, u) \end{aligned}$$

If we repeat this with the other two upward pointing panels, we compute the points $f(B, u)$ and $f(C, u)$. We can now repeat the process with these three points as they differ only in one argument and this argument has the values A, B, C, giving us the point $f(u, u) = F(u)$.

Pictorially, we have



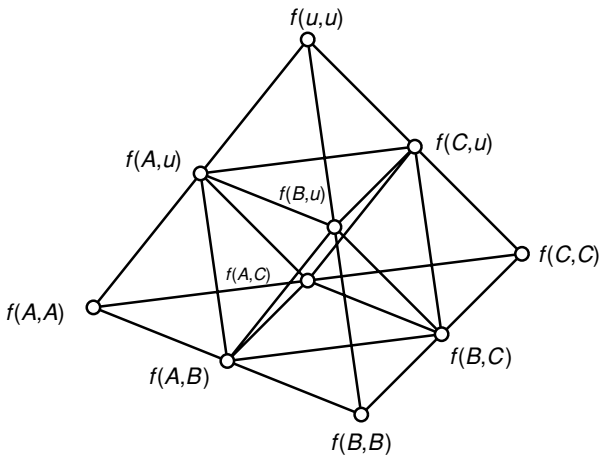
We can see from the labeling of the points that this de Casteljau algorithm gives us subdivision:



Thus, if we evaluate at the center of our domain, we divide our triangular domain into three subtriangles, each drawn with a different shade in the figure above. Then in our range, the points connected to the triangles with the same shade are the control points for the Bézier patch associated with the corresponding shaded domain.

However, while this subdivision algorithm is a nice generalization of the subdivision algorithm for curves, it has serious problems. Because we subdivide in the interior, our subdomains (and corresponding Bézier patches) have poor aspect ratios. Further issues with this subdivision method will be discussed in Section 4.2.3.

We can also look at a dataflow diagram of the evaluation. Now, instead of a triangle dataflow diagram, we will have a tetrahedral dataflow diagram:



The initial control points are at the bottom of the diagram, with the evaluation proceeding upward.

For curves, we noted that the subdivision points lay on the outer edges of the dataflow triangle. Now for surfaces, we note that the subdivision points lie on the outer faces of the dataflow tetrahedron.

4.1.2 Exercise

1. State and prove the multiaffine blossoming principle for a degree n polynomial with domain of arbitrary dimension d.

4.1.3 Derivatives

For curves, we looked at the multilinear blossom to find derivatives. But look more closely at the multiaffine blossom we have created for surfaces: If we build the blossom from the barycentric form of the Bernstein polynomials, then it's already a multilinear blossom. All that remains to get the multilinear blossom is to remove the restriction that the u_i sum to one (where the u_i are the barycentric coordinates of our domain point u).

However, we still do not have the statement about the derivatives of F. We will make one now. First note that we have to work with directional derivatives since our domain is more complicated than in the curve case. Thus, our statement will look like

$$F^{(j)}_{v_1,\ldots,v_j}(u) = \frac{n!}{(n-j)!} f\left(u^{\langle n-j\rangle}, v_1, \ldots, v_j\right). \tag{4.1}$$

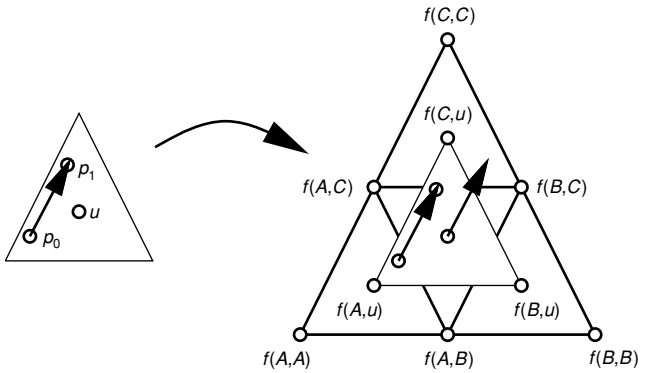
Notationally, we have several problems: first, we need to subscript blossom arguments to distinguish between them, and second, we need to subscript the barycentric coordinates of a point. Also, we need a notation for vectors in space and a "vector" as a tuple (e.g., as in $\vec{i} = (i_0, \ldots, i_d)$ where i_js are integer). And we want to overload $|v|$ notation: does it mean the length of a vector or does it mean the sum of the barycentric coordinates. All of these notational issues are readily overcome, but they do complicate things.

We will not prove Eq. (4.1) here; Ramshaw discusses it in his research report [18].

So what does the above formula mean geometrically? Consider the first derivative in a direction $v = p_1 - p_0$ where p_1 and p_0 are points in the domain a unit distance apart:

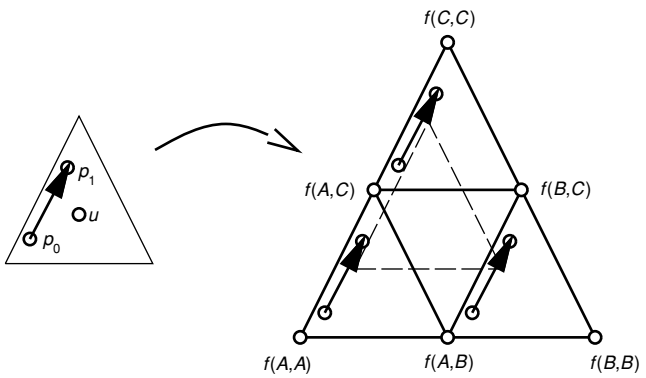
$$\begin{aligned} F'_v(u) &= nf\left(u^{\langle n-1\rangle}, v\right) \\ &= nf\left(u^{\langle n-1\rangle}, p_1 - p_0\right) \\ &= n\left[f\left(u^{\langle n-1\rangle}, p_1\right) - f\left(u^{\langle n-1\rangle}, p_0\right)\right] \end{aligned}$$

One way to picture this is we evaluate at u for all our parameters but one, then we map our domain direction to the remaining triangle:



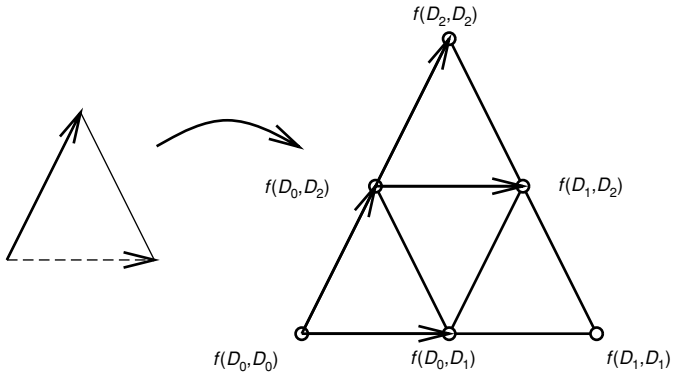
In the range, I have drawn the image of the domain vector $p_1 - p_0$ twice. The left occurrence was computed as the difference of $F(p_1) - F(p_0)$. I have then translated this vector to start at $F(u)$, since it is a directional derivative for that point.

Alternatively, we can evaluate at v first. This gives us a Bézier "surface" that we can evaluate anywhere to find the derivative at a point:



For surface patches we are often interested in the tangent plane behavior along a boundary. The tangent plane behavior is completely defined if we know the first derivative in two non-parallel directions at every point along the boundary curve. One direction is easy: we just take the derivative of the boundary curve. Note that this is the directional derivative of the surface along one of the edges of the domain triangle. For the other direction, we commonly choose a

second direction in the domain triangle:



In this figure, the solid vector in the domain on the left is our boundary direction, and the dashed line is the crossboundary direction. In the range on the right, the two solid vectors are linear interpolated to compute the boundary derivative, while the dashed vectors are linearly blended to compute the crossboundary derivative.

What about higher order derivatives? We could evaluate the blossom to find these derivatives. However, just like for curves, we treat the first derivative function as a Bézier function, and apply what we know about first derivatives to it.

Note in particular what this says about the mixed partial derivative at a patch corner in two directions parallel to the edges:

$$\begin{aligned} F^{(2)}_{v_1,v_2}(A) &= n(n-1)f\left(u^{\langle n-2\rangle}, v_1, v_2\right) \\ &= n(n-1)\left[f\left(u^{\langle n-2\rangle}, v_1, C\right) - f\left(u^{\langle n-2\rangle}, v_1, A\right)\right] \\ &= n(n-1)\left(\left[f\left(u^{\langle n-2\rangle}, B, C\right) - f\left(u^{\langle n-2\rangle}, A, C\right)\right]\right. \\ &\quad \left.-\left[f\left(u^{\langle n-2\rangle}, B, A\right) - f\left(u^{\langle n-2\rangle}, A, A\right)\right]\right) \end{aligned}$$

Thus, to compute the mixed partial at a corner point in two edge directions, first compute the two vectors used for the first derivative in one direction; the difference between these vectors gives the mixed partial derivative.

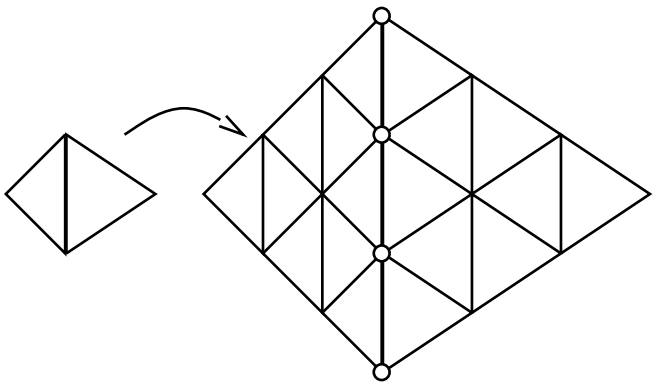
The mixed partial is often called the "twist vector." There are four vertices involved in computing the twist vector. Suppose that these four vertices lie on the corners of a parallelogram. Then the twist vectors will be zero. Thus, the twist vector measures how far out from a parallelogram these four vertices are. It is called "twist" because it is measured the way in which the crossboundary derivative twists out of the tangent plane as you move away from the corners.

4.1.4 Parametric Continuity

Now that we know about derivatives, what about continuity? There are two types of continuities commonly discussed for surfaces: parametric continuity and geometric continuity. The latter is more interesting for geometric modeling, but it is beyond the scope of this lecture. Here, we will just consider parametric continuity.

Once we start talking about two patches meeting with parametric continuity, however, we have to be a bit careful: the barycentric coordinates hides much of the detail needed for derivatives. In particular, we have to associate the two domains of our patches. Typically, we will just make them two triangles that intersect in an edge. It is along this common edge that the two surfaces will meet.

It is easy to get C^0 continuity: since the boundary of each surface is a Bézier curve, we merely insist that the two patches share boundary control points:



For higher order continuity, we state without proof the following proposition quoted from Ramshaw's tech report [18].

Proposition Let A and B be two adjacent regions in the parameter plane P that meet along the segment $[s, t]$ of the line L in P, where $s \neq t$. The two n-ic polynomial surfaces $F_A : A \rightarrow Q$ and $F_B : B \rightarrow Q$ join with C^k continuity along the segment $[s, t]$ if and only if their multiaffine blossoms f_A and f_B agree on all argument bags that include at least $(n - k)$ points anywhere along L.

We are interested in the geometry of these conditions. The C^0 conditions are simple: the boundary control points of the two patches must be equal. For illustration, assume we are working with two quadratic patches F and G with blossoms f and g, parameterized over domain triangles $\triangle ABC$ and $\triangle CBD$, respectively, with $D = d_a A + d_b B + d_c C$ and $A = a_d D + a_b B + a_c C$.

Then the C^0 continuity conditions are

$$f(B, B) = g(B, B)$$
$$f(B, C) = g(B, C)$$
$$f(C, C) = g(C, C)$$

That is, if all the arguments to the blossom come from $\{B, C\}$, then the blossoms f and g must be equal.

For C^1 continuity, we need to have C^0 continuity, and further, Ramshaw's theorem states that if all but one of the arguments to the blossom come from $\{B, C\}$, then the blossoms f and g must be equal. Clearly, for these patches to meet with C^1 continuity, the following equations must be true:

$$f(A, B) = g(A, B)$$
$$f(A, C) = g(A, C)$$

Further, if these five equalities hold (three for C^0 continuity and two for C^1 continuity), then f and g will be equal when one or more of the blossom arguments is B or C. This is seen since

$$\begin{aligned} f(t, B) &= t_a f(A, B) + t_b f(B, B) + t_c f(C, B) \\ &= t_a g(A, B) + t_b g(B, B) + t_c g(C, B) \\ &= g(t, B) \end{aligned}$$

where t_a, t_b, t_c are the barycentric coordinates of t relative to $\triangle ABC$. A similar argument shows that $f(t, C) = g(t, C)$. Thus, these five conditions are sufficient to ensure C^1 continuity between F and G along their common boundary.

Figure 4.2 (left) illustrates the C^1 continuity conditions for these patches to meet with C^1 continuity. Note that as a result of these conditions, pairs of adjacent panels must be coplanar, although this is only a necessary condition for C^1 (i.e., the stronger condition that these panels are affine images of the panels formed by the two domain triangles must hold for C^1 continuity).

For C^2 continuity, the C^1 conditions must hold. Further, for our quadratic example, the blossoms must be equal when none of the arguments are from $\{B, C\}$. The conditions on our control points in this example are the following: we construct

$$f(A, D) = d_a f(A, A) + d_b f(A, B) + d_c f(A, C)$$

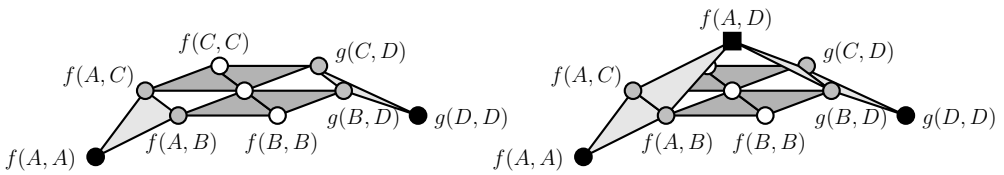


FIGURE 4.2: Two quadratic Bézier patches meeting with C^1 and C^2 continuity

and

$$g(A, D) = a_d f(D, D) + a_b f(D, B) + a_c f(D, C)$$

For patches F and G to meet with C^2 continuity, we must have $f(A, D) = g(A, D)$. A similar argument to the one above shows that this is a strong enough condition to ensure that $f(t, u) = g(t, u)$ for all t, u. Figure 4.2 (right) illustrates the C^2 continuity conditions.

Note that the situation of two quadratics meeting with C^2 continuity is an unusual special case, since if two quadratic patches meet with C^2 continuity, then they are different parameterizations of the same degree 2 polynomial and thus they must agree for any set of arguments; however, normally we would not expect the blossoms of two neighboring patches to agree unless at least one of their arguments came from $\{B, C\}$.

If we increase the degree of the patches, we get similar conditions, but on more panels between the adjacent patches. Further, if we consider conditions for higher continuity between the patches, we see conditions similar to the C^1 and C^2 conditions: for C^{2k+1} continuity, there is a condition of coplanar panels, and for C^{2k} continuity, there is a condition of coincident points.

For a further discussion of these continuity conditions, see [11, 15].

4.1.5 Surfaces Above the Plane

Suppose that we want a bivariate polynomial function, $z = F(x, y)$, in Bézier form. For curves, we spaced the x value of the control points equally in the domain. For surfaces we will do a similar thing: if our domain triangle is $\triangle ABC$ and our desired function is to be of degree n, then we place the control point $P_{\vec{i}} = P_{i_0 i_1 i_2}$ at $((i_0/n)\,A + (i_1/n)\,B + (i_2/n)\,C, z_{\vec{i}})$, where $(i_0/n)\,A + (i_1/n)\,B + (i_2/n)\,C$ represents both the x and y coordinates of the point. The $z_{\vec{i}}$ may be placed anywhere.

Note that our C^1 constraints simplify for a function over the plane. Due to the uniform x–y spacing of the control points, it is easy to see that planarity of a boundary panel implies it is an affine image of the domain.

4.1.6 Exercise

1. For a polynomial $z = F(x, y)$ with a domain triangle $\triangle ABC$, to find the Bézier representation we set the xy-coordinates of the Bézier control point P_{i_0,i_1,i_2} to $(i_0/n)\,A + (i_1/n)\,B + (i_2/n)\,C$. Prove that this setting of the xy-coordinates yields the identity mapping for the xy-coordinates, i.e., show that for t in the xy-plane,

$$t = \sum_{\vec{i},\, |\vec{i}|=n} B^n_{\vec{i}}(t)((i_0/n)\,A + (i_1/n)\,B + (i_2/n)\,C)$$

where $\vec{i} = (i_0, i_1, i_2)$.

4.1.7 Storing the Control Points

How do we store and access the control points for a triangular Bézier patch? A naive approach would be to store them in an $n \times n \times n$ array indexed by i_0, i_1, and i_2. But since the three integers in the index are not independent, we can store them in an $n \times n$ array, indexing on i_0 and i_1. However, we will still be wasting half of our storage space. If this wasted space is not a concern, then this method is probably the best method. If it is a concern, we can use a better method: we can store them in a linear array.

To see how to store the triangular array in a linear array, we make the following observation: if $\mathrm{Dim}(n, d)$ is the number of control points in a degree-n, dimension-d control net, then

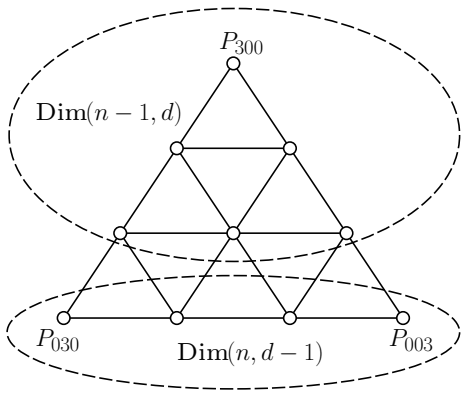
$$\mathrm{Dim}(n, d) = \mathrm{Dim}(n-1, d) + \mathrm{Dim}(n, d-1).$$

As base cases, if either the dimension or degree are zero, then the value of Dim is 1. (If either d or n are negative, then the value of Dim is 0.) Note that $\mathrm{Dim}(n, d) = \binom{n+d}{d}$ is the dimension of the space of polynomials in d variables of degree less than or equal to n.

Why would we compute Dim using the recursive calls rather than directly use the combinatorial formula? Because 13! exceeds the precision of 32-bit integers. Thus, for high degree and/or high dimension objects, we have to use an alternative method of computation.

In practice, we would use table lookup to get the values of Dim (a 20×20 table is not very big, although $\mathrm{Dim}(18, 18)$ will overflow a 32-bit integer; $\mathrm{Dim}(17, 17)$ is a 32-bit value, but even this is probably bigger than you need, since a control point array that size would not fit in memory).

For surfaces, we are working with the case of $d = 2$, giving the following pictorial representation for this formula:



We will store the $\mathrm{Dim}(n-1, d)$ part in the initial part of a linear array, and the Dim $(n, d-1)$ part immediately after it. So our first control point should be P_{n00}. And our last

control point should be P_{00n}. In fact, the entire set of control points whose first index is 0 are the $\mathrm{Dim}(n, d-1)$ part in the above picture. Our algorithm to compute the location of a point P_i will proceed by looking at the first index and computing the size of the $\mathrm{Dim}(n - i_0, d)$ part. This gives us the location of the "control net" of one lower dimension that contains our point. Thus, we repeat this process for each of the lower dimensions, as illustrated by the following code:

```
sum = n
offset = 0
for j = 0 to d − 1 do
                    sum = sum − i_j
                    offset += Dim(sum −1, d − j)
endfor
return offset
```

4.1.8 Efficient Evaluation at a Single Point

We would like to evaluate our triangular Bézier patches efficiently. If we store the control points in a 2D array using the first two indices of their multiindex, then the following is a reasonable evaluation method for one step of de Casteljau's algorithm:

```
for i = 1 to n do
    for j = 0 to n − i do
        P[i][j] = w0 P[i][j] + w1 P[i + 1][j] + w2 P[i][j + 1]
    end
end
```

Suppose, however, that we wish to store our control points in a linear array. One way to evaluate one level of de Casteljau's algorithm (combining control points from an array `CP` and storing the results in a separate array `ECP`) is the following:

```
for i = n − 1 downto 1 do
    for j = n − i − 1 downto 0 do
        k = n − i − j − 1
        ECP[MtoI(i, j, k)] = w0CP[MtoI(i + 1, j, k)] +
            w1CP[MtoI(i, j + 1, k)] + w2CP[MtoI(i, j, k + 1)]
    end
end
```

In the code, MtoI( ) is a procedure for converting a multiindex into a linear index.

But this is inefficient: for each evaluation we have to run the MtoI( ) code four times. Generally, it is most efficient to store the control points in reverse lexicographical order (i.e, for a cubic, (3,0,0), (2,1,0), (2,0,1), (1,2,0), (1,1,1), (1,0,2), (0,3,0), (0,2,1), (0,1,2), (0,0,3)). If we store them in this fashion, then

$$\text{MtoI}(i, j, k) = \text{MtoI}(i + 1, j, k)$$

and

$$\text{MtoI}(i, j, k + 1) = \text{MtoI}(i, j + 1, k) + 1$$

Using one more trick, we can use the following code to evaluate one level of de Casteljau's algorithm in place:

```
i = 0
j = 1
for l = 0 to n − 1 do
    for m = 0 to l do
        CP[i] = w0CP[i] + w1CP[j] + w2CP[j + 1]
        i++
        j++
    end
    j++
end
```

4.2 FAST EVALUATION ON A GRID OF POINTS

To render a surface, we commonly construct a piecewise linear approximation to the surface and render it using standard Z-buffer graphics hardware. Earlier, we looked at how to efficiently evaluate a patch for a single point and normal on the surface. In this section, we will look at various methods for evaluating at a set of point appropriate for triangulation.

Much of the material in this section is based on [6] and [17]. See also [14] for further discussion on surface evaluation.

4.2.1 A Grid of Evaluation Points

To draw a Bézier patch, we will evaluate it on a "grid" of domain points, triangulate the evaluation points, and draw the triangles.

At each domain point, we need to compute the barycentric coordinates. Since we want a grid of samples, by exploiting the grid property, we can compute the barycentric coordinates of

each domain point without much effort:

```
for i = 0 to S do
    for j = 0 to S − i do
        k = S − i − j
        u0 = i/S
        u1 = j/S
        u2 = k/S
        P = EvalPatch(T,u0,u1,u2)
    end
end
```

When we evaluate, we will want to evaluate for both position and normal. We also need to put these evaluation points in an array, and then generate triangles from these points to give to OpenGL for rendering.

The question of what value to use for S remains. Typically, just using $S = 10$ should be sufficient.

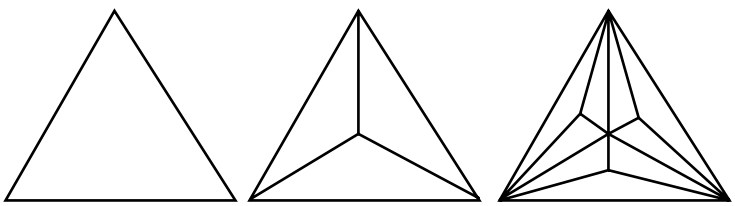
4.2.2 Implementations

1. Write an evaluator for triangular Bézier patches. Extend this evaluator to a tesselator (e.g., chop it up into a set of triangles). Your tesselations should have both position and normals for each sample point.

4.2.3 3-to-1 Subdivision

To make an adaptive scheme that avoids the "what is S" question, we might try extending the de Casteljau subdivision of curves method to triangular patches. The idea is that we would repeatedly perform 3-to-1 subdivision until the control points are nearly coplanar.

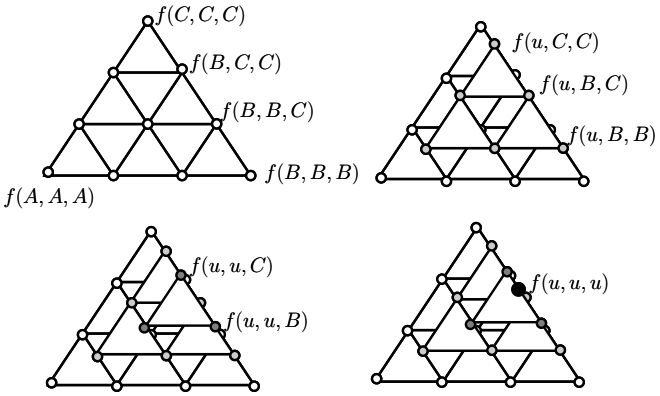
There are two problems with 3-to-1 subdivision. The first is that the aspect ratio of the polygons is terrible. For a variety of reasons, we prefer that the triangles are close to equilateral. However, 3-to-1 subdivision gives triangles that get longer and skinnier each time we subdivide:



The second (and fatal) flaw of 3-to-1 subdivision (for rendering purposes) is that it does not divide the edges of the patches, and we would never end up with a set of coplanar control points.

4.2.4 2-to-1 Subdivision

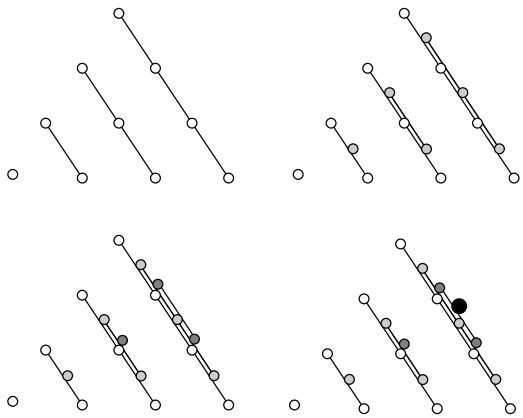
A 2-to-1 subdivision is an efficient way to subdivide a triangular Bézier patch. Consider what happens when we evaluate on the boundary of the domain triangle:



The barycentric coordinates of the domain point of evaluation are $(0, u_b, u_c)$. Thus, in each affine combination of the algorithm, we have

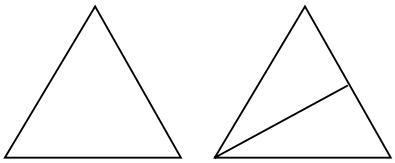
$$f(u, X, Y) = 0 f(A, X, Y) + u_b f(B, X, Y) + u_c f(C, X, Y) = u_b f(B, X, Y) + u_c f(C, X, Y)$$

where X, Y have the values u, A, B, or C depending on which triangle we are combining. Essentially, we are evaluating (in the cubic case) four curves: a constant curve, a linear curve, a quadratic curves, and a cubic curve:

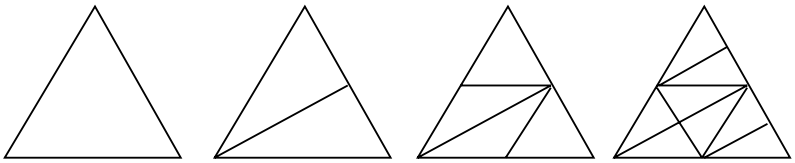


Further, if we look at the 3-to-1 subdivision produced by de Casteljau's algorithm, one of the domain subtriangles is degenerate: it is a line. Thus, we are really performing 2-to-1 subdivision, and we can compute it using only repeated curve evaluation. The decrease is cost is significant: 10 pairwise affine combinations versus 10 affine combinations of three points. This is two-thirds the number of multiplications and half the number of additions.

At first glance, the aspect ratio appears to get worse when we do 2-to-1 subdivision:



But if we repeatedly perform 2-to-1 subdivision by connecting from the previous subdivision point to the two edges that had not been subdivided at the previous step, after three steps, we see that all triangles have the same aspect ratio as the triangles occurring after the first step:

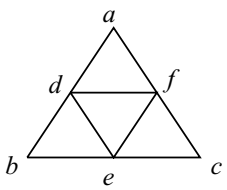


So we see that although there is a worsening of the aspect ratio in the first 2-to-1 subdivision step (and possibly in the second step), the aspect ratio gets no worse than this.

4.2.5 4-to-1 Subdivision

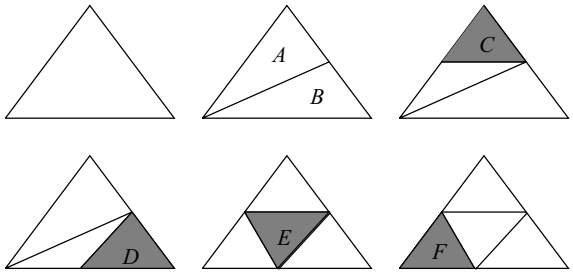
We saw earlier that if we evaluate a triangular Bézier patch with de Casteljau's algorithm, then we perform a 3-to-1 subdivision of the patch. Note, however, that we will never subdivide the edges of the patch (or of the subpatches) if we repeat the subdivision step.

A nicer subdivision technique would be to perform a 4-to-1 subdivision by splitting all three edges of the domain and connecting in the obvious fashion:



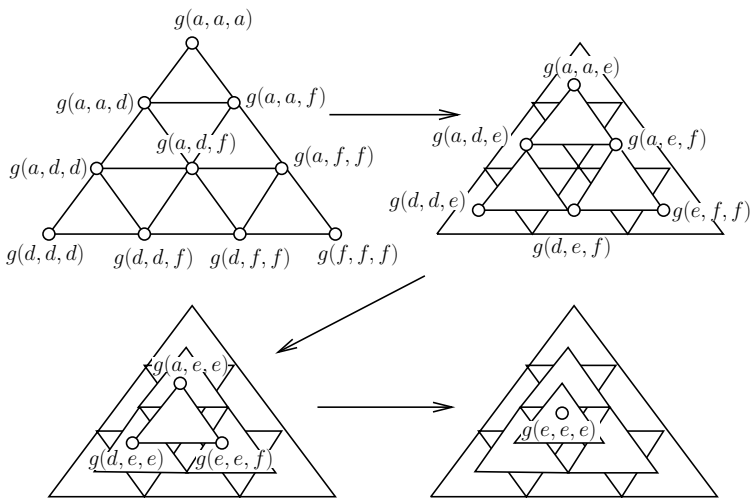
Thus, our original domain is $\triangle abc$ and we want to subdivide it into four subtriangles, $\triangle\ adf$, $\triangle\ bed$, $\triangle\ cfe$, and $\triangle\ def$. The question is, how do we perform such a subdivision on the triangular Bézier patch?

One answer is to perform repeated subdivision on the patch. Most of the subdivisions are 2-to-1 subdivisions, where we perform 3-to-1 subdivision on an edge and ignore the degenerate patch (or more efficiently, using curve subdivision).



Here, we start with our original patch and subdivide it at f. We then subdivide patch A at d giving us patch C. We subdivide B at e to get D. Next, we "subdivide" either C at e or D at d to get E. Finally, we "subdivide" E at b to get patch F.

Note that in evaluating patch C at e (i.e., at a point outside of the domain triangle for C), we really do get the control points of patch E from the de Casteljau evaluation. In the figure below, the control points of E are along the bottom edge of each of the four diagrams (i.e., the evaluations of $g(\,)$ where all arguments are a mix of d, e, or f).



There is also a way you can subdivide using only four evaluations (Exercise 1 of Section 4.2.8) [5].

4.2.6 Curve Evaluation

Finally, we note that typically we want to tessellate the surface for rendering. This is commonly done by evaluating the surface for position and normal at a uniform grid of points, and creating a mesh of triangles from these points. This requires evaluating at a uniform sampling on each boundary of the patch. Since the boundary of a triangular Bézier patch is a Bézier curve, rather than perform a surface evaluation for each of these boundary samples, we can instead evaluate the boundary control points as a Bézier curve to obtain the boundary samples (we will also need to evaluate the second layer of control points as a Bézier curve to obtain a crossboundary derivative to compute the surface normal).

This suggests another algorithm for tessellating a triangular Bézier patch over a regular grid: step along in one barycentric direction. At each step, perform a 2-to-1 subdivision of the triangular Bézier patch. Take either of the resulting subpatches and sample the split boundary repeatedly in the other two barycentric directions to obtain samples along that boundary.

Pseudo-code for the algorithm appears in Fig. 4.3. The code computes $S + 1$ samples along each edge, and stores the results in a 2D array P, with half the space in the array unused. Figure 4.4 illustrates the key ideas of the algorithm, showing the two 2–1 subdivisions of the patch F, and showing which control points are extracted to be evaluated as curves, with the *white* control points used to determine the position and one tangent direction on the surface, and the *gray* control points used to compute a second tangent direction on the surface, allowing us to compute the surface normal. Note that the inner `for` loop evaluates the curves at uniform step sizes; thus, the curve evaluations in this `for` loop could be implemented using forward differencing for additional speed improvements.

```
for i=0 to S-1
    u = i/S
    (P1,P2) = TwoToOneSubdivision(F,u,0,1-u)
    (Q1,Q2) = TwoToOneSubdivision(P1,u,1-u,0)
    (W,G)=ExtractTwoLayersCurve(Q1)
    for j=0 to S-i
        v = j/(S-i)
        P[i][j] = EvalDoubleCurve(W,G,v)
    end
end
P[S][0] = F[n][0][0]
```

FIGURE 4.3: Pseudo-code for regular sampling of S per side using two 2–1 subdivisions and curve evaluation. "W" and "G" refer to the white and gray control points of Fig. 4.4

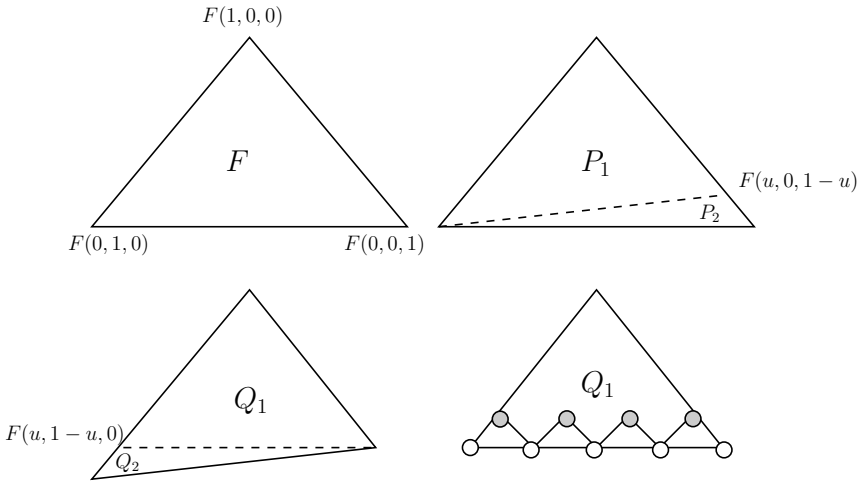


FIGURE 4.4: Two 2–1 subdivisions, followed by curve evaluation

4.2.7 Cracking Problems

For 4-to-1 subdivision and 2-to-1 subdivision, we can imagine an adaptive algorithm that checks if a patch's control points are coplanar or close to coplanar. If they are, then the patch is drawn as a triangle formed by the corner control points. If the patch control points are not close to coplanar, then we subdivide the patch and repeat the process (doing two 2-to-1 subdivision steps if we are using 2-to-1 subdivision).

Such an algorithm has the advantage that the piecewise linear approximation will converge to the surface with close to a minimum number of triangles for the required tolerance. However, this algorithm suffers from a serious problem. If we have adjacent patches that are subdivided to different levels (Fig. 4.5, left), then a crack may form along their common boundary. Although the gap may be small (less than a pixel in width), because of numerical issues, when these triangles are rendered we may get pixel drop out along the common edge (Fig. 4.5, middle and right; in these figures, the "triangle" on the lower right has an additional vertex on the common

FIGURE 4.5: A crack may form between adjacent patches subdivided to different levels

edge). This problem usually precludes using this type of adaptive algorithm, or necessitates additional processing to "insert the missing vertices" to avoid pixel drop out.

4.2.8 Discussion

Each of the evaluation algorithms for triangular patches discussed in this section has its uses. For renderering purposes, unless speed is a critical issue, it is likely that the simplest algorithm of using de Casteljau's algorithm to evaluate on a grid of points (Section 4.2.1) is the best choice as it is simple to implement.

If speed is the primary concern, then the 2-to-1 subdivision algorithm together with forward differencing for curve evaluation (Sections 4.2.4 and 4.2.6) will probably give the fastest results, although if different levels of subdivision are used for different patches, then crack resolving issues may dominate run-time costs. The 3-to-1 and 4-to-1 subdivision algorithms are less useful for rendering, but both algorithms at times come in useful for specialized applications such as change of basis.

4.2.9 Exercise

1. Above, we saw how to perform 4–1 subdivision of a triangular Bézier patch using five de Casteljau evaluations. Show how to obtain 4–1 subdivision using only four de Casteljau evaluations.

4.3 TENSOR-PRODUCT SURFACE PATCHES

A second and more commonly used type of surface patch is a rectilinear surface patch called a *tensor-product patch*. One advantage of this type of surface patch is that it can be used in either Bézier form or B-spline form, the latter surface being piecewise polynomial with moderately high continuity. One disadvantage of the tensor-product form is that we are limited in the topological type of surfaces we can easily model.

Mathematically, a tensor-product Bézier surface is given as

$$T(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} B_i^n(u) B_j^m(v)$$

where u, v is a rectilinear domain space and the Bs are the Bernstein polynomials. Note that this describes a polynomial function of degree $n + m$. Also note that we can bracket this expression as follows:

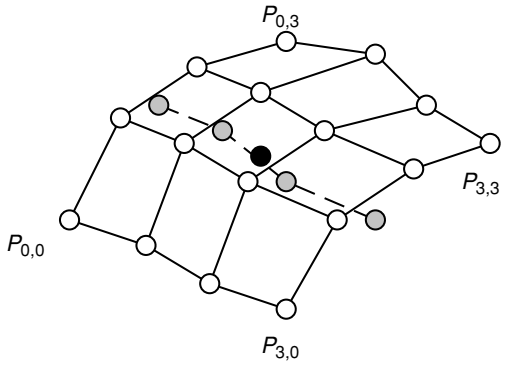
$$T(u, v) = \sum_{i=0}^{n} \left\{ \sum_{j=0}^{m} P_{i,j} B_j^m(v) \right\} B_i^n(u)$$

For a fixed i, the expression inside the curly braces is a curve. Evaluating each such curve at v gives us a new expression,

$$T(u, v) = \sum_{i=0}^{n} Q_i B_i^n(u) \tag{4.2}$$

where $Q_i = \sum_{j=0}^{m} P_{i,j} B_j^m(v)$. We again recognize this as a curve, and evaluating this last curve gives us our point on the surface.

Thus, we can evaluate a tensor-product surface by performing repeated curve evaluation.



Note also that we could have rearranged the terms to evaluate the $B_i^n(u)$s first:

$$T(u, v) = \sum_{j=0}^{m} \left\{ \sum_{i=0}^{n} P_{i,j} B_i^n(u) \right\} B_j^m(v)$$

There appears to be a need for such a second evaluation: if we want to compute the normal to the surface as well as the point $T(u, v)$ then we need to find two partial derivatives on the surface. A de Casteljau evaluation of (4.2) will give us one of the partial derivatives (use the points generated in the next to last step of the algorithm); to get the other partial, we can evaluate the rearranged tensor-product surface, and the last de Casteljau evaluation will give us the other partial derivative. While this method gives us both the position and normal to the surface, more efficient methods of evaluating tensor-product surfaces are discussed in Section 4.4.

4.3.1 The Blossom of a Tensor-Product Surface

When we blossom a tensor-product surface, we blossom independently in each of its parameter directions. Thus, the blossom of $T(u, v)$ is

$$t(u_1, \ldots, u_n; v_1, \ldots, v_m)$$

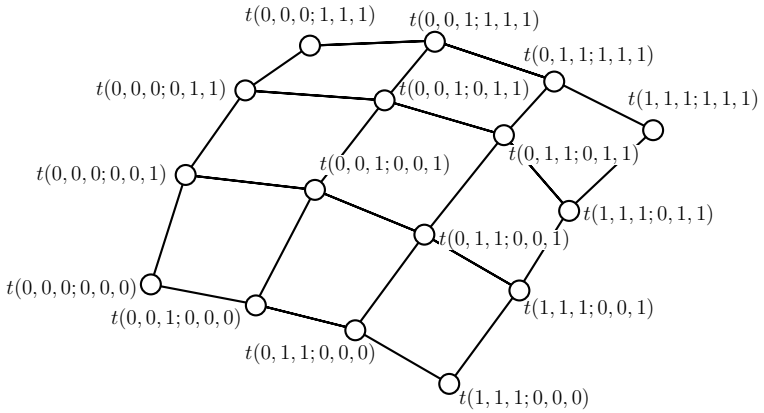


FIGURE 4.6: A bicubic surface with blossom labels on its control points

This function is multiaffine in both the u and v parameters, and it is symmetric in the u parameters, and symmetric in the v parameters, but we cannot interchange u and v parameters.

Suppose we try to evaluate t at u, v. Then we get the following:

$$\begin{aligned} t\big(u^{\langle n\rangle}; v^{\langle m\rangle}\big) &= (1-u)t\big(0, u^{n-1}; v^{\langle m\rangle}\big) + ut\big(1, u^{n-1}; v^{\langle m\rangle}\big) \\ &\vdots \\ &= \sum_{i=0}^{n} B_i^n(u)t\big(0^{\langle n-i\rangle}, 1^{\langle i\rangle}; v^{\langle m\rangle}\big) \\ &= \sum_{j=0}^{m}\sum_{i=0}^{n} B_i^n(u)B_j^m(v)t\big(0^{\langle n-i\rangle}, 1^{\langle i\rangle}; 0^{\langle m-j\rangle}, 1^{\langle j\rangle}\big) \end{aligned}$$

and again we see that the control points are defined by "nice" blossom values (Fig. 4.6).

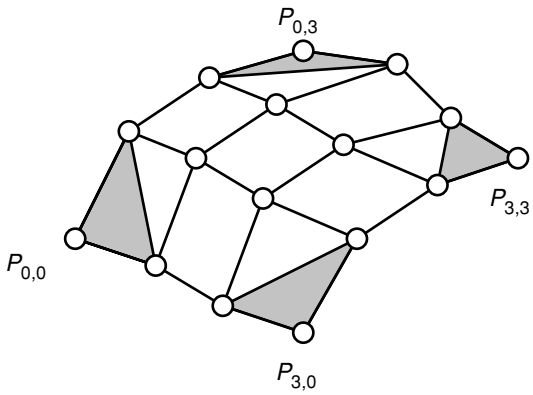
4.3.2 Derivatives

The partial derivatives with respect to the parameter variables of a tensor-product surface can be obtained by taking the derivatives with respect to the appropriate curves. Thus, if we want the partial with respect to u at u_0, v_0, then we evaluate the blossom at $t(u_0^{\langle n-1\rangle}, \delta; v_0^{\langle m\rangle})$. More precisely,

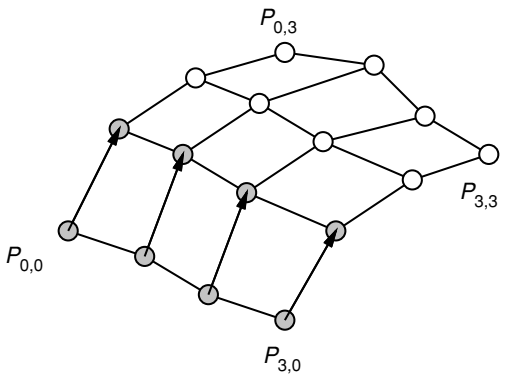
$$\frac{\partial T(u_0, v_0)}{\partial u} = nt\big(u_0^{\langle n-1\rangle}, \delta; v_0^{\langle m\rangle}\big)$$

We compute partials with respect to v and higher order derivatives in a similar fashion.

Again note that this means the corner point and the two points adjacent to it span the tangent plane as shown by the shaded panels in the following figure:



Also, the partial derivative with respect to u along the $v = 0$ (or $v = 1$) boundary is given by the difference of the first two (last two) layers of control points:



This follows from wanting to compute

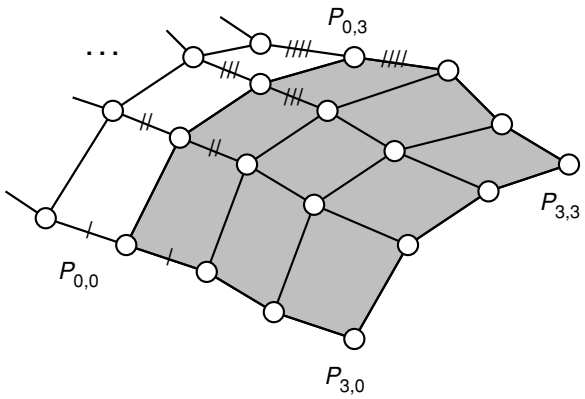
$$\frac{\partial T(0, v)}{\partial u} = nt\left(0^{\langle n-1\rangle}, \delta; v^{\langle m\rangle}\right)$$

When we evaluate the first argument set $n - 1$ times at 0, we are left with just two rows of control points. Evaluating at δ leaves us with a blossom whose coefficients are the vectors formed by the pairwise difference in these control points.

4.3.3 Continuity

Parametric continuity for tensor-product patches is much simpler than for triangular patches: We just use curve continuity. Thus, if we want two patches to meet with C^1 continuity along a

boundary, we just construct the curves in that parametric direction to meet with C^1 continuity:



4.3.4 Tensor-Product B-Splines

Since continuity is defined by the curves, we could use B-spline curves instead of Bézier curves, giving us tensor-product B-splines:

$$T(u, v) = \sum_{i,j}^{n,m} P_{i,j} N_i^n(u) N_j^m(v)$$

Note, however, that each of the B-spline curves in a parametric direction must have the same knot vector. Thus, there are a total of two knot vectors for a tensor-product B-spline: one in the u direction and one in the v direction.

Further realize that while the tensor product B-spline surface has polynomial degree $n + m$, the continuity across the boundaries is C^{n-1} and C^{m-1} depending on in which parameteric direction you are moving. In particular, note that a bicubic surface is only C^2.

However, while the continuity across patches is not as high as we might hope (given the degree of the surface), the main advantage of tensor-product B-spline surfaces is that no modeling work had to be done to achieve this continuity. The user just positions the control points, and the patches automatically meet with a reasonable level of smoothness. It is this "automatic smoothness" that makes tensor product B-splines useful in geometric modeling. Their main weaknesses are difficulties in using them to model nonrectangular patches and in using them to contruct surfaces that do not have a nice rectilinear topology.

4.3.5 Surfaces Above the Plane

Tensor-product Bézier surfaces above the plane are simple to construct: just place your curves along constant, uniformly separated u lines, with control points spaced uniform v distance apart along each curve.

Tensor-product B-spline surfaces with uniform knot vectors can also be constructed over the plane by uniform placement of the control points, although their positioning is a bit trickier. Things are complicated further for nonuniform knot sequences.

4.3.6 Generalizing the Dimension

If we wish to generalize to higher dimensional domains, we merely add another parametric parameter (e.g., $T(u, v, w) = \sum P_{i,j,k} N_i(u) N_j(v) N_k(w)$). Free form deformations are an example of a method that use trivariate tensor-product volumes [21].

4.3.7 Storage

Unlike triangular patches, we do not have to do anything special to efficiently store our control points: we merely store them in an $n \times m$ array.

4.4 ALTERNATIVE EVALUATION METHODS FOR TENSOR PRODUCT SURFACES

At the start of this chapter, we saw how to evaluate a tensor-product surface using repeated curve evaluation, evaluating first the rows of the control net, and then evaluating one column of control points. This evaluation gave us a point on the surface and a partial derivative. To obtain the other partial derivative, we needed to evaluate the surface a second time, first evaluating the columns, and then evaluating a single row. This double evaluation of the surface is inefficient. In this section, I discuss a variety of other methods for evaluating tensor-product surfaces.

4.4.1 Repeated Bilinear Interpolation

The idea in repeated bilinear interpolation is that a degree 1×1 tensor-product surface is a bilinear surface equal to its own blossom: $T(u, v) = t(u; v)$. We can linearly weight the control points on the rows with u to obtain two points that we linearly weight with v, or vice versa (Fig. 4.7). Further, the normal to the surface is

$$N(u, v) = (t(u; 1) - t(u; 0)) \times (t(1; v) - t(0; v))$$

For a higher degree $n \times n$ tensor-product surface, we note that in each "panel" of control points, the blossom values that vary have the same pattern as the bilinear blossom labels. For example, referring to Fig. 4.8, for the four control points on the shaded panel, the first two

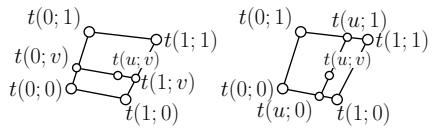


FIGURE 4.7: Evaluating a bilinear blossom

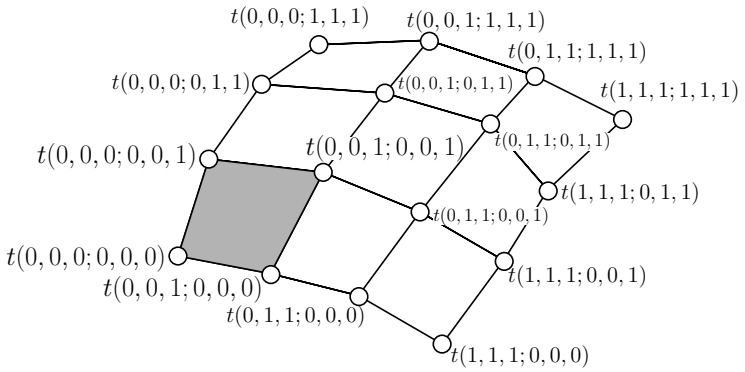


FIGURE 4.8: The nonfixed blossom arguments of each panel of control points form the argument bag of a bilinear function

arguments in the u set of arguments are 0,0, and the first two arguments in the v set of arguments are 0,0. The remaining argument in each parameter can be evaluated like a bilinear patch.

Such an arrangement holds for each panel of the surface. If we evaluate each of the $(n-1)\times(n-1)$ panels of a $n\times n$ surface, we get a new set of $(n-2)\times(n-2)$ panels. The blossom arguments to these panels have the same property: we may evaluate them as bilinear patches. We can repeat this process until we obtain a single panel, which we can evaluate for both the position and normal to the surface. This process is illustrated for bicubic patches in Fig. 4.9.

The bilinear interpolation algorithm requires fewer computations than does the repeated curve evaluation algorithm. It is often made a bit more efficient by noticing that

$$\begin{aligned}&(1-u)[(1-v)P_{0,0}+vP_{0,1}]+u[(1-v)P_{1,0}+vP_{1,1}]\\&=(1-u)(1-v)P_{0,0}+(1-u)vP_{0,1}+u(1-v)P_{1,0}+uvP_{1,1}\end{aligned}$$

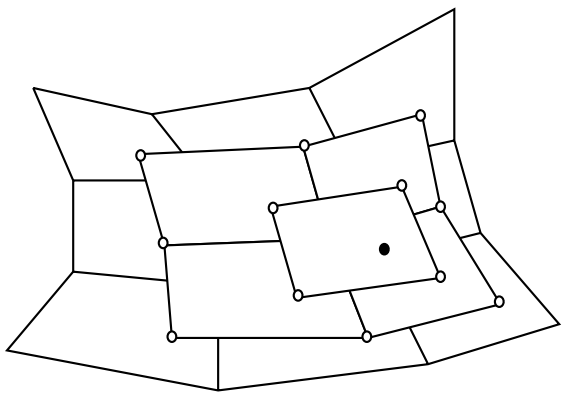


FIGURE 4.9: Repeated bilinear interpolation of a bicubic surface

where the four weights $(1-u)(1-v)$, $(1-u)v$, $u(1-v)$, uv can be computed once at the start of the algorithm. This algorithm suffers from the problem that it only works on bi-n-ic surfaces; e.g., it would fail on a 3×4 tensor-product surface.

4.4.2 Repeated Curve Evaluation: Revisited

The repeated curve evaluation algorithm can be modified to be more efficient at computing both the position and normal to the surface. The main idea here is that, starting with the blossom $t(u_1, \ldots, u_n; v_1, \ldots, v_m)$, we want to obtain $t(u^{\langle n-1\rangle}, u_1; v^{\langle m-1\rangle}, v_1)$ as efficiently as possible, leaving us with control points

$$t\big(u^{\langle n-1\rangle}, 0; v^{\langle m-1\rangle}, 0\big),\ t\big(u^{\langle n-1\rangle}, 1; v^{\langle m-1\rangle}, 0\big)$$
$$t\big(u^{\langle n-1\rangle}, 0; v^{\langle m-1\rangle}, 1\big),\ t\big(u^{\langle n-1\rangle}, 1; v^{\langle m-1\rangle}, 1\big)$$

This bilinear panel can then be evaluated for both position and normal.

The idea of Mann–DeRose [16] was to evaluate curves in the parametric direction of lower degree (i.e., if $n < m$, then evaluate in u first), but stop each curve evaluation one short of completion. This leaves you with two degree m curves having control points

$$t\big(u^{\langle n-1\rangle}, 0; 0^{\langle m\rangle}\big),\ t\big(u^{\langle n-1\rangle}, 0; 0^{\langle m-1\rangle}, 1\big), \ldots, t\big(u^{\langle n-1\rangle}, 0; 1^{\langle m\rangle}\big)$$

and

$$t\big(u^{\langle n-1\rangle}, 1; 0^{\langle m\rangle}\big),\ t\big(u^{\langle n-1\rangle}, 1; 0^{\langle m-1\rangle}, 1\big), \ldots, t\big(u^{\langle n-1\rangle}, 1; 1^{\langle m\rangle}\big).$$

We then evaluate these two curves one short of completion, leaving the desired bilinear panel.

The algorithm is illustrated in Fig. 4.10. On the upper left, we start with a grid of control points. On the upper right, we have run de Casteljau's algorithm on each row, stopping one short of completion. This leaves two columns to evaluate, which we have done in the lower left. Finally, in the lower right, we have connected the remaining four points, and labeled them with the blossom values.

4.4.3 Recursive Subdivision

Another algorithm we might consider for evaluating a tensor-product surface is recursive subdivision. If we evaluate a tensor-product patch on its interior, by retaining some of the intermediate de Casteljau points (essentially, the ones that are retained for curve subdivision), we can subdivide our patch into four subpatches. This is easily verified by checking the blossom values on those control points.

We can now envision an algorithm that recursively subdivides the tensor-product patch until the control points of each piece are coplanar to within some tolerance, at which point we can approximate the patch with two triangles or even a quadrilateral. However, as with the 4-to-1 recursive subdivision algorithm for triangular Bézier patches, this algorithm suffers

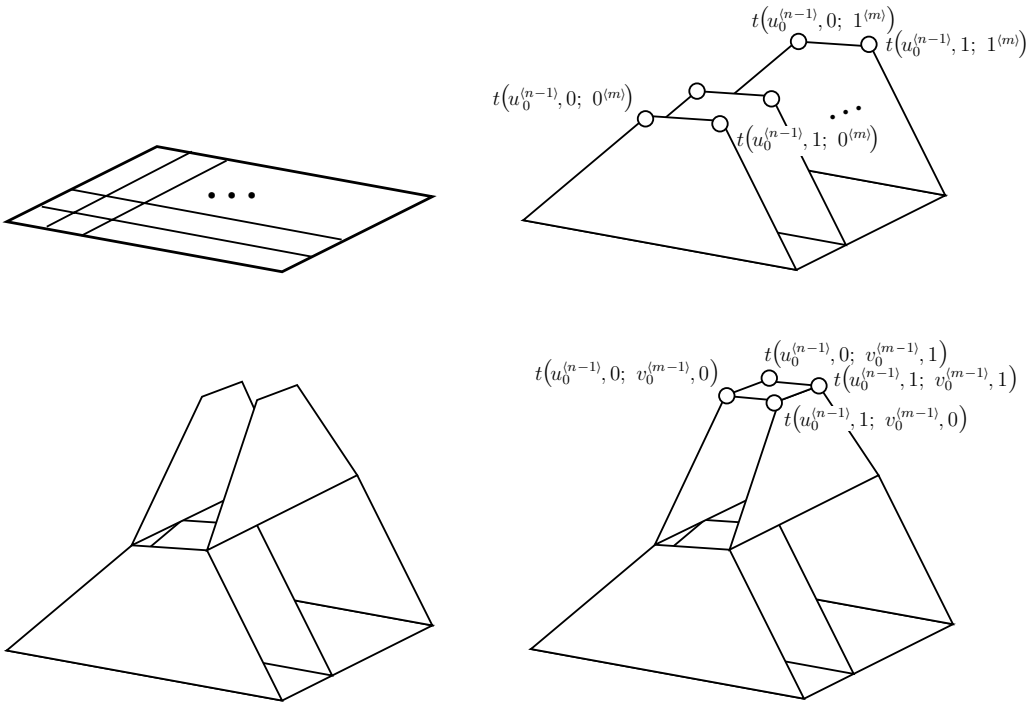


FIGURE 4.10: Illustration of Mann–DeRose algorithm

from a "cracking" problem, where different levels of subdivision within a patch results in cracks appearing on the tessellation.

4.4.4 Curve Evaluation

As with triangular Bézier patches, to draw our surface we typically wish to evaluate over a grid of points at uniform samples. In this case, we can evaluate in one parametric direction at u_0, giving us a curve. We then evaluate this curve at a sequence of v values, giving us all of our sample points $T(u_0; v_0)$, $T(u_0; v_1), \ldots, T(u_0; v_S)$. We then repeat this process with successive u samples until we have sampled our surface on a uniform domain grid. Again, when we sample in the u direction, we retain two layers of control points and perform two curve evaluations in the v direction to obtain surface normals. Forward differencing methods may be used to further accelerate the process [13].

4.4.5 Discussion

If speed is not an issue, for rendering a tensor-product surface the simple execution of repeated curve evaluation twice (introduction of Section 4.3) would likely suffice. However, the repeated bilinear interpolation algorithm (if the degrees in the two parameteric directions are equal) or

the repeated curve evaluation algorithm (Section 4.4) are no harder to implement and both are significantly faster, so either one would make a better choice then twice performing repeated curve evalution.

For speed, the curve evaluation method at the end of Section 4.4 using forward differencing will likely be the fastest option. As always, if patches are tessellated at different level of granularity, then stitching the pieces together to avoid cracks will dominate the execution time.

4.4.6 Exercise

1. Draw a repeated bilinear interpolation figure similar to Fig. 4.9 but for a degree 3×4 patch to see why repeated bilinear interpolation fails if $n \neq m$.

Bibliography

[1] Phillip Barry and Ron Goldman. Algorithms for progressive curves. In Goldman and Lyche, editors, *Knot Insertion and Deletion Algorithms for B-Spline Curves and Surfaces*. SIAM, Philadelphia, PA, 1993, pp. 11–63.

[2] Phillip Barry and Ron Goldman. Factored knot insertion. In Goldman and Lyche, editors, *Knot Insertion and Deletion Algorithms for B-Spline Curves and Surfaces*. SIAM, Philadelphia, PA, 1993, pp. 65–88.

[3] Phillip Barry and Ron Goldman. Knot insertion algorithms. In Goldman and Lyche, editors, *Knot Insertion and Deletion Algorithms for B-Spline Curves and Surfaces*. SIAM, Philadelphia, PA, 1993, pp. 89–133.

[4] Richard Bartels and John Beatty. A technique for the direct manipulation of splines. In *Graphics Interface*. Morgan-Kaufmann, Palo Alto, CA, 1989, pp. 33–39.

[5] Wolfgang Böhm. Subdividing multivariate splines. *Computer-Aided Design* 15(6):345–352, November 1983.

[6] Jeromy Carriere. Evaluating tensor product and triangular Bézier surfaces. *Technical Report CS-95-22*, University of Waterloo, Ontario, Canada, May 1995.

[7] Gerald Farin. *Curves and Surfaces for CAGD*, 5th edn. Morgan-Kaufmann, San Francisco, 2002.

[8] Barry Fowler and Richard Bartels. Constraint-based curve manipulation. *IEEE Computer Graphics and Applications* 13(5):43–49, September 1993.

[9] Jean Gallier. *Curves and Surfaces in Geometric Modeling: Theory and Algorithms*. The Morgan Kaufmann Series in Computer Graphics. Morgan-Kauffman, San Francisco, 1999.

[10] Ron Goldman. *Pyramind Algorithms: A Dynamic Programming Approach to Curves and Surfaces for Geometric Modeling*. Morgan-Kaufmann, San Francisco, 2003.

[11] Ming-Jun Lai. Geometric interpretation of smoothness conditions of triangular polynomial patches. *Computer Aided Geometric Design* 14(2):191–199, 1997.doi:10.1016/S0167-8396(96)00028-3

[12] J. Lane and R. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *EEE Transactions on Pattern Analysis and Machine Intelligence* 2:35–46, 1980.

[13] Sheue-Ling Lien, Michael Shantz, and Vaughan Pratt. Adaptive forward differencing for rendering curves and surfaces. In *Proceedings of SIGGRAPH 1987*, Vol. 21, 1987, pp. 111–118.doi:org/full_text

[14] Suresh Lodha and Ron Goldman. A unified apporach to evaluation algorithms for multivariate polynomials. *Mathematics of Computation* 66(220):1521–1553, 1997. doi:10.1090/S0025-5718-97-00862-4

[15] Stephen Mann. Adjusting control points to achieve continuity. *Computer Aided Geometric Design* 19:589–602, 2002.doi:10.1016/S0167-8396(02)00147-4

[16] Stephen Mann and Tony DeRose. Computing values and derivatives of Bézier and B-spline tensor products. *Computer Aided Geometric Design* 12(1):107–110, February 1995.doi:10.1016/0167-8396(94)00030-V

[17] Jorg Peters. Evaluation and approximate evaluation of multivariate Bernstein form on a regularly partitioned simplex. *ACM Transactions on Mathematical Software* 20(4):460–480, December 1994.doi:10.1145/198429.198434

[18] Lyle Ramshaw. Blossoming: A connect the dots approach to splines. *Technical Report 19*, DEC SRC, June 1987.

[19] Lyle Ramshaw. Blossoms are polar forms. *Computer Aided Geometric Design* 6(4):323–358, November 1989.

[20] Lyle Ramshaw. On multiplying points: The paired algebras of forms and sites. *Technical Report 169*, DEC SRC, May 2001.

[21] Thomas Sederberg and Scott Parry. Free-form deformation of solid geometric models. In *Proceedings of SIGGRAPH*. ACM, New York, 1986, pp. 51–160.

[22] Hans-Peter Seidel. An introduction to polar forms. *IEEE Computer Graphics and Applications* 13(1):38–46, January/February 1993.doi:10.1109/38.180116

[23] Etienne Vouga and Ron Goldman. Two blossoming proofs of the Lane–Riesenfeld algorithm. In *Proceedings of Dagstuhl Seminar 05221*. Dagstuhl, Germany, in press.

Index

Biography

Stephen Mann is an Associate Professor in the David R. Cheriton School of Computer Science and cross-appointed to the Mechanical Engineering Department at the University of Waterloo, Waterloo, Ontario, Canada. He received a B.A. in computer science and pure mathematics at the University of California, Berkeley, and has a Masters in Computer Science and Ph.D. in Computer Science and Engineering from the University of Washington in Seattle. His research interests include CAGD, geometric modeling, computer graphics, and the mathematical foundations of computer graphics.