Programming Support for Blossoming
The Blossom Classes

by

Wayne Liu

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1995

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below and give address and date.

Abstract

A C++ library has been created to facilitate prototyping of curve and surface modeling techniques. The library provides general-purpose blossoming datatypes to support creation of modeling techniques based on blossoming analysis. The datatypes have efficient operations which are generalizations of important CAGD algorithms, and can be used to implement many algorithms. Most importantly, the library is able to inter-operate with user-supplied datatypes or routines to create complex modeling techniques.

Acknowledgements

I thank NSERC and ITRC for their financial assistance.

I thank the following people, without whom this thesis would not have been possible: my supervisor, Stephen Mann, for his many good counsels; my readers, Richard Bartels and Hans-Peter Seidel; the members of the Computer Graphics Lab, especially Anne Jenson, Rob Kroeger, Greg Veres, Fabrice Jaubert, who were always willing to help when I got stuck; my "family" in the Laymen's Fellowship, for their prayers and encouragement; my parents, for their support and motivation; my Lord and Saviour Jesus Christ, for His love and daily provision.

Contents

List of Figures

Chapter 1

Introduction

Computer Aided Geometric Design (CAGD) is concerned with modeling curves and surfaces on computers. Research focuses on finding various techniques of representing curves and surfaces in computer-compatible form, and algorithms for manipulating these representations. This research has applications in CAD/CAM. For a general introduction to CAGD, see Farin's book [11].

The most successful techniques represent curves and surfaces with piecewise polynomial functions. Some examples of these are Bézier patches and NURBS. Many properties of such techniques are most easily studied using blossoming analysis. Blossoming analysis was introduced into CAGD in 1987 [16, 7]. Since that time, it has proven to be a simple and powerful mathematical tool. It does not require advanced mathematical concepts, yet it reveals the properties of important modeling techniques. Researchers continue to apply blossoming analysis to find new modeling techniques that have useful properties.

In addition to analyzing new ideas mathematically, researchers must also implement prototypes, computer programs that test the practicality of these ideas. Thus, programming is an essential step in CAGD research.

The task of programming involves translating from the mathematical analysis into computer code. This translation is often difficult. The problem lies in translating mathematical concepts, such as piecewise polynomials, into computer language concepts, such as floating-point arithmetic. It is unclear how to translate from one to the other, and in general, they bear no resemblance to each other. Ideally, the programmer should be able to manipulate the same concepts in the code as in the analysis. Then, the translation process would be straight-forward, and the programming would be simple. The solution is to create datatypes for blossoming.

A datatype is simply an abstract set of objects with operations that can be performed on these objects. In this case, the objects correspond to concepts used in the blossoming analysis, such as blossoms, tensors, bases, spaces or multi-indices. The operations perform meaningful actions on the objects in terms of blossoming analysis. Thus, the programmer can use the operations to manipulate the mathematical concepts in the code.

In this thesis, I have developed the Blossom Classes, a C++ library to support programming with blossoming datatypes. The library is designed to be useful for many applications. The library provides a set of general datatypes that can be used to code many modeling techniques. In creating operations for the datatypes, I discovered generalizations of important CAGD algorithms. The resulting operations are efficient building blocks for many algorithms.

The outstanding feature of the library is its ability to work with other datatypes and tools. This feature allows the library to be used for coding complex modeling techniques that require combining datatypes and routines from other libraries.

Overview of Chapters

In Chapter 2, I review the technique of blossoming analysis. Note that for reasons discussed in that chapter, this thesis uses an alternative approach to blossoming, based on the tensor construction.

In Chapter 3, I describe previous work on programming support for blossoming, list the requirements for such systems to be effective, and describe the design of the Blossom Classes in light of these requirements.

In Chapter 4, I derive the algorithms that compute the datatypes' operations. Then, I analyze their run time requirements and their potential for accumulating round-off errors.

In Chapter 5, I discuss the implementation of the library, with particular reference to how the library works with user-supplied datatypes.

In Chapter 6, I evaluate the usefulness of the library by using it to implement different techniques and algorithms.

In Chapter 7, I summarize the results of this work and list further work.

Chapter 2

Background on Blossoming Analysis

This chapter reviews concepts of blossoming analysis, following the development given by Ramshaw [16]. I give less rigorous versions of Ramshaw's definitions, and I omit the proofs. Note that both Ramshaw and this thesis use a different formulation of blossoming analysis than in the general literature. See Section 2.2 for a discussion of the difference.

2.1 Geometric Spaces

The natural framework in which to analyze CAGD techniques is *affine geometry*. (DeRose [10] gives a thorough introduction to affine geometry.) An *affine space* is made up of a set of points and an associated linear space of vectors. The points have the operation of vector addition. Subtraction of two points is defined as $x - y \equiv \vec{v}$, where $\vec{v}$ is the unique vector such that $x = y + \vec{v}$. An *affine combination* of points $x_0, \ldots, x_n$, with scalars $\alpha_0, \ldots, \alpha_n$, $\sum_{k=0}^{n} \alpha_i = 1$, is the point

$$\sum_{k=0}^{n} \alpha_i x_i \equiv x_0 + \sum_{k=1}^{n} \alpha_i (x_i - x_0).$$

A *basis* of an affine space is a set of points $\{x_0, \ldots, x_n\}$ such that $\{x_1 - x_0, \ldots, x_n - x_0\}$ is a basis of the associated linear space. This implies every point, x, can be expressed in a unique way as an affine combination of $x_0, \ldots, x_n$, that is, $x = \sum_{k=0}^{n} \alpha_i x_i$. The α_i are called the *coordinates* of x with respect to the basis $\{x_0, \ldots, x_n\}$. An affine space is n-dimensional if its associated linear space is n-dimensional. This implies a basis of an affine space has $n + 1$ elements.

Since the derivatives of curves and surfaces are vectors, the analysis of CAGD techniques would be simplified if points and vectors can be treated in the same way. The standard technique is to embed both points and vectors in a $n+1$ dimensional linear space, called the *linearized space* of the affine space. The construction proceeds as follows: add a new element $\vec{v}_0$ to the associated (n-dimensional) linear space, and identify a point x_0 with this element.

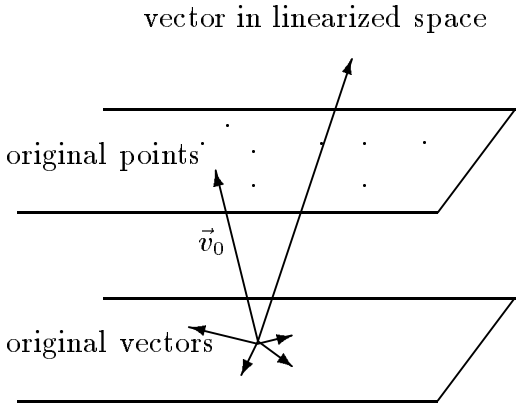


Figure 2.1: Construction of linearized space

The $n+1$ dimensional linear space is made of elements of the form $\alpha\vec{v}_0 + \vec{v}$ for some scalar α and vector $\vec{v}$ in the original linear space. Figure 2.1 shows this construction.

The α is called the *flavor* (or weight or mass) of the element. By construction, all points x are represented as $x = x_0 + (x - x_0) = \vec{v}_0 + (x - x_0)$ and all vectors are represented as $0\vec{v}_0 + \vec{v}$; thus a point has flavor 1 and a vector has flavor 0. In this thesis, an arbitrary element of the linearized space is written as x, that is, when it doesn't matter whether the element is a point or vector. When the element is known to be a vector, it is written as $\vec{v}$. When the element is known to be a point, it is still written as x, but it will be called a point.

Two kinds of bases are commonly used for linearized spaces. The first is obtained from a basis of the affine space (i.e. only points); this kind of basis is called a *simplex*. The second is obtained from a basis of the original linear space with $\vec{v}_0$ added to it; this kind of basis is called a *frame*. Coordinates of vectors in a linearized space are called *homogeneous coordinates*.

A *projective* space is obtained from a linearized space by taking all elements of flavor $\alpha \neq 0$ and dividing by α: $1\vec{v}_0 + (x - x_0)/\alpha$. This division is called a *projection*.

These geometric spaces are useful for analyzing polynomials: ordinary polynomials have domains that are affine; *homogeneous* polynomials have domains that are linear; *rational* polynomials have *ranges* that are projective spaces.

2.2 Polynomials and Blossoms

Blossoming

The standard approach to blossoming is based on the *multi-affine blossom*, which is a symmetric and multi-affine (affine in each argument) map of n arguments. The following theorem states that polynomials and multi-affine blossoms are essentially the same.

Theorem 2.2.1 (The Blossoming Principle) For every polynomial, $F : X \to Y$, of degree n, there is exactly one multi-affine blossom, $\hat{f} : X^n \to Y$, such that

$$F(u) = \hat{f}(\underbrace{u, \ldots, u}_{n}),$$

and vice versa.

If $\hat{f}$ is a multi-affine blossom of three arguments, then the symmetric property implies $\hat{f}(x, y, z) = \hat{f}(y, x, z) = \hat{f}(\text{any permutation of } x, y, z)$. The multi-affine property implies

$$\hat{f}(\alpha w + (1 - \alpha)x, y, z) = \alpha \hat{f}(w, y, z) + (1 - \alpha)\hat{f}(x, y, z).$$

The symmetric and multi-affine properties allow $\hat{f}$ to be computed from known values of $\hat{f}$ at given arguments. For example, as illustrated in Figure 2.2, given the values of $\hat{f}(0, 1, 2)$, $\hat{f}(1, 2, 3)$, $\hat{f}(2, 3, 4)$, and $\hat{f}(3, 4, 6)$, it is possible to compute $\hat{f}(2.7, 1.8, 3.4)$. First note that by the symmetric property,

$$\hat{f}(0, 1, 2) = \hat{f}(1, 2, \underline{0}).$$

Then, by the multi-affine property,

$$\hat{f}(1, 2, \underline{2.7}) = \underline{.1}\hat{f}(1, 2, 0) + \underline{.9}\hat{f}(1, 2, 3).$$

Thus, from the first pair of values of $\hat{f}$, the new value $\hat{f}(1, 2, \underline{2.7})$ can be computed. In a similar manner, using the next pair and the last pair, $\hat{f}(2, 3, \underline{2.7})$ and $\hat{f}(3, 4, \underline{2.7})$ can be computed. These three new values combine to give $\hat{f}(2, \underline{2.7}, \underline{1.8})$ and $\hat{f}(3, \underline{2.7}, \underline{1.8})$, which in turn combine to give $\hat{f}(\underline{2.7}, \underline{1.8}, \underline{3.4})$.

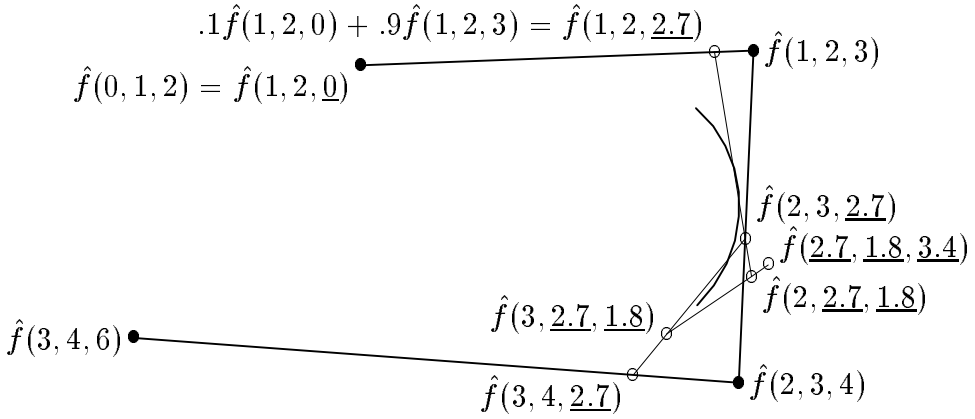


Figure 2.2: Evaluating a blossom

This previous example evaluated the blossom of a curve. Since the concept of blossom is independent of dimension, the same approach can evaluate surfaces. For example, as

illustrated in Figure 2.3, given the planar points, $a_0, a_1, b_0, b_1, c_0, c_1, x$, and the blossom values $\hat{f}(a_0, a_1)$, $\hat{f}(a_0, b_0)$, $\hat{f}(b_0, b_1)$, $\hat{f}(b_0, c_0)$, $\hat{f}(c_0, c_1)$, $\hat{f}(c_0, a_0)$, it is possible to compute $\hat{f}(x, x)$. First rewrite $\hat{f}(c_0, a_0)$ as $\hat{f}(a_0, c_0)$ using the symmetric property. Then use the multi-affine property to write

$$\hat{f}(a_0, \underline{x}) = \lambda_0 \hat{f}(a_0, a_1) + \lambda_1 \hat{f}(a_0, b_0) + \lambda_2 \hat{f}(a_0, c_0),$$

where the λs are the barycentric coordinates of x in triangle a_1, b_0, c_0. In the same way, the values of $\hat{f}(b_0, x)$,$\hat{f}(c_0, x)$, and finally $\hat{f}(x, x)$ are computed.

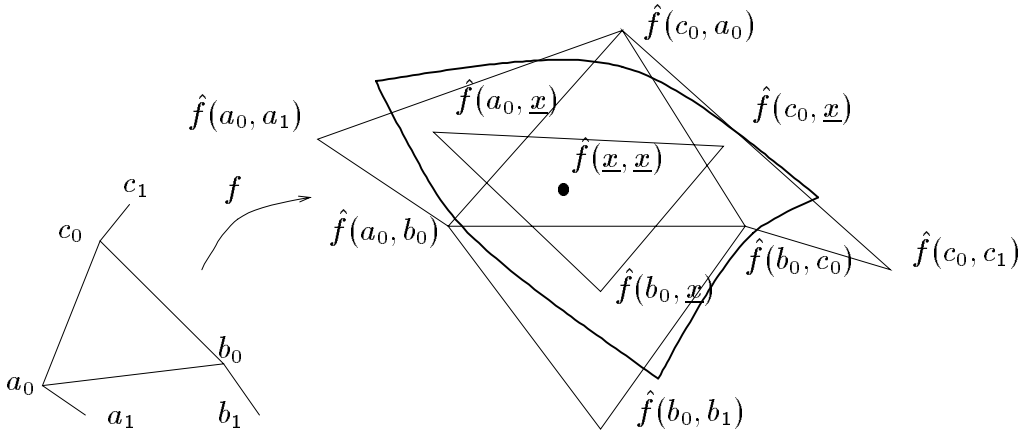


Figure 2.3: Evaluating a surface blossom

Figure 2.4 shows these two computations in another way, showing which blossom values combine to calculate new values. These diagrams are known as "wonderful triangles" [12].

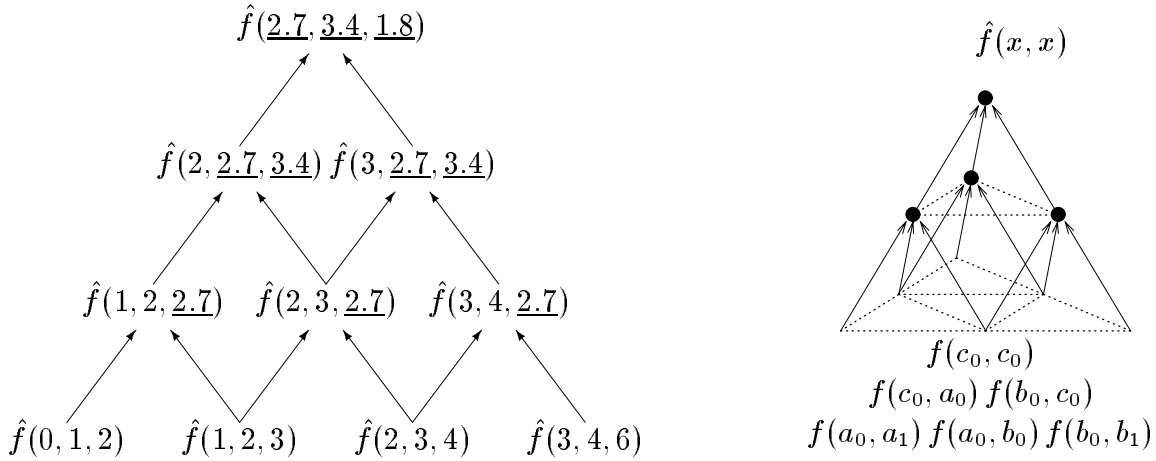


Figure 2.4: Evaluation computations

The arguments of the known values of the blossom must follow a pattern to allow new blossom values to be computed. This pattern is defined in Section 2.5.

Tensoring

Another way to evaluate $\hat{f}(2.7, 3.4, 1.8)$ is to perform the previous computation "backwards", and find the "weight" of each blossom value:

$$\begin{aligned}
\hat{f}(2.7, 3.4, 1.8) &= .3\hat{f}(2, 2.7, 3.4) - .7\hat{f}(3, 3.4, 1.8) \\
&= -.06\hat{f}(1, 2, 2.7) + .57\hat{f}(2, 3, 2.7) + .49\hat{f}(3, 4, 2.7) \\
&= -.024\hat{f}(0, 1, 2) + .382\hat{f}(1, 2, 3) + .6665\hat{f}(2, 3, 4) + .0245\hat{f}(3, 4, 6)
\end{aligned}$$

The data flow is shown in Figure 2.5.

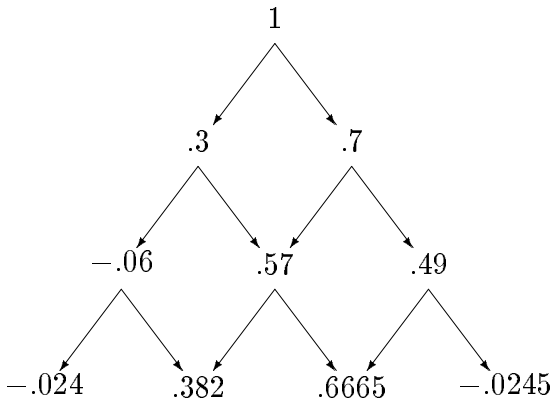


Figure 2.5: Getting the weights of blossom values

Note that the weights $(-.024, .382, .6665, .0245)$ *do not depend on the blossom* $\hat{f}$, but only on the arguments to the blossom. This observation suggest that it is worthwhile to study arguments to a blossom. The tensor construction allows blossom arguments to be studied as mathematical entities.

The construction for *symmetric n-tensors* (or *tensor* for short) proceeds recursively. A point, x, is a 1-tensor. An $(n + m)$-tensor can be constructed from an m-tensor, u, and an n-tensor, v, by *tensor multiplication*, to get the tensor $u \otimes v$. Finally, affine combinations of n-tensors, $\sum_i \alpha_i u_i$, are n-tensors.

Tensor multiplication is associative, commutative and distributes over affine combinations: $(u \otimes v) \otimes w = u \otimes (v \otimes w)$, $u \otimes v = v \otimes u$, and $v \otimes \sum_i \alpha_i u_i = \sum_i \alpha_i v \otimes u_i$.

Since the multiplication is commutative, the standard notation for symmetric tensors omits the $\otimes$ symbol for tensor multiplication. Thus, $u \otimes v$ is simply uv. From now on, this thesis uses the shorter notation. Another simplification is to denote an n-tensor, u, multiplied by itself k times as u^k.

An n-tensor obtained by multiplying n 1-tensors together is called a *simple tensor*: $u = x_1 \ldots x_n$ (i.e., written without a summation sign, as opposed to $.2x_1x_2 + .8x_3x_4$). Simple tensors are equivalent to blossom arguments.

By construction, n-tensors form an affine space; the points of this space are n-tensors, while the vectors are combinations of n-tensors, $\sum_i \alpha_i u_i$, where the scalars sum to 0. This space is the *nth symmetric tensor space*, denoted $X^{\otimes n}$.

The following theorem states that *affine* maps from the tensor space are equivalent to multi-affine blossoms. Such an affine map is called an *affine blossom.*

Theorem 2.2.2 (The Tensoring Principle) For every multi-affine blossom, $\hat{f} : X^n \to Y$ of degree n there is exactly one affine blossom, $f : X^{\otimes n} \to Y$, such that $\hat{f}(x_1, \ldots, x_n) = f(x_1 \cdots x_n)$, and vice versa.

As a corollary, this theorem says affine blossoms are equivalent to polynomials. The affine blossom f corresponding to a polynomial F is called the *blossom of* F.

Denote by $vX^{\otimes n-m}$ the subset of the space of n-tensors obtained by multiplying the m-tensor v by $(n-m)$-tensors. This subset forms a subspace of $X^{\otimes n}$ isomorphic to $X^{\otimes n-m}$. A basis of this subspace can be obtained by multiplying v to a basis of $X^{\otimes n-m}$. Let $f|_v$ denote the blossom f restricted to this subspace. Then $f|_v$ can be considered a blossom over $X^{\otimes n-m}$, where $f|_v(u) = f(vu)$. $f|_v$ is called the *partial evaluation of* f *at* v.

The computation of weights can now be restated using tensor notation. First, to distinguish between the *point* 2.7 in the 1-D domain from the *scalar* 2.7, the point is written $\mathbf{2.7}$. Now, using the commutativity and distributivity of tensor multiplication, the simple tensor $\mathbf{2.7\,3.4\,1.8}$ can be written as an affine combination of the simple tensors $\mathbf{0\,1\,2}$, $\mathbf{1\,2\,3}$, $\mathbf{2\,3\,4}$, and $\mathbf{3\,4\,6}$:

$$\begin{aligned}
\mathbf{2.7\,3.4\,1.8} &= (.3\,\mathbf{2} + .7\,\mathbf{3})\,\mathbf{2.7\,3.4} = .3\,\mathbf{2\,2.7\,3.4} - .7\,\mathbf{3\,3.4\,1.8} \\
&= -.06\,\mathbf{1\,2\,2.7} + .57\,\mathbf{2\,3\,2.7} + .49\,\mathbf{3\,4\,2.7} \\
&= -.024\,\mathbf{0\,1\,2} + .382\,\mathbf{1\,2\,3} + .6665\,\mathbf{2\,3\,4} + .0245\,\mathbf{3\,4\,6}.
\end{aligned}$$

An affine blossom is simply an affine map over the tensors. Thus,

$$\begin{aligned}
f(\mathbf{2.7\,3.4\,1.8}) &= f(-.024\,\mathbf{0\,1\,2} + .382\,\mathbf{1\,2\,3} + .6665\,\mathbf{2\,3\,4} + .0245\,\mathbf{3\,4\,6}) \\
&= -.024 f(\mathbf{0\,1\,2}) + .382 f(\mathbf{1\,2\,3}) + .6665 f(\mathbf{2\,3\,4}) + .0245 f(\mathbf{3\,4\,6}).
\end{aligned}$$

Linearizing

The concepts of blossoming can be extended to linearized spaces to make tensors from both vectors and points. This extension allows derivatives of polynomials to be computed using blossoms.

Similar to the affine case, *multi-linear blossoms* are defined as symmetric, multi-linear maps. The only added restriction is that the multi-linear blossom must be flavor multiplicative (mapping points with flavors α_1 to α_n to a point with flavor $\alpha_1 \cdots \alpha_n$).

The construction of *linearized tensor spaces* proceeds as in the affine case, but using linear combinations instead of affine combinations. The tensor space is a linearized space.

The *flavor* of a tensor is defined as follows: the flavor of a simple tensor is defined to be the product of the flavors of its components; the flavor of a sum of simple tensors is the sum of the flavors of each simple tensor. A *linear blossom* is a linear map on $X^{\otimes n}$. The notation for a linearized tensor is the same as for an affine tensor, except that 1-tensors that are vectors are written as $\vec{v}$ rather than just x.

Linear blossoms are useful for obtaining derivative information.

Theorem 2.2.3 The directional derivative of F at x in the direction $\vec{v}$ is $\frac{\partial}{\partial \vec{v}} F(x) = nf(x^{n-1}\vec{v})$.

The theorem can be restated as "the blossom of $\frac{\partial}{\partial \vec{v}} F$ is $nf|_{\vec{v}}$". The theorem can be recursively applied to $f|_{\vec{v}}$ as a $n-1$ blossom to get a general formula:

$$\begin{aligned} \frac{\partial^k}{\partial \vec{v}_1 \cdots \partial \vec{v}_k} F(x) &= n(n-1)\cdots(n-k+1) f(x^{n-k}\vec{v}_1 \cdots \vec{v}_k) \\ &= n(n-1)\cdots(n-k+1) f|_{\vec{v}_1 \cdots \vec{v}_k}(x^{n-k}) \\ &= n(n-1)\cdots(n-k+1) f|_{x^{n-k}}(\vec{v}_1 \cdots \vec{v}_k). \end{aligned} \tag{2.1}$$

This theorem is important for piecewise-polynomial methods since it states that two polynomials, F and G, join C^k at x if and only if $f|_{x^{n-k}} = g|_{x^{n-k}}$.

An example will clarify the concept of linear tensors. Suppose the following values of the blossoms f and g are given: $f(\mathbf{0\,1\,2})$, $f(\mathbf{1\,2\,3}) = g(\mathbf{1\,2\,3})$, $f(\mathbf{2\,3\,4}) = g(\mathbf{2\,3\,4})$, $f(\mathbf{3\,4\,6}) = g(\mathbf{3\,4\,6})$, and $g(\mathbf{4\,6\,6})$. Equation 2.1 can be used to show that $F(x)$ and $G(x)$ meet C^2 at 3.

Let $\vec{\delta}$ be the unit vector pointing in the direction of increasing x. For example, $\vec{\delta} = \mathbf{3} - \mathbf{2}$. Then the theorem implies

$$\begin{aligned} \frac{d^2}{dx^2} F(3) &= 6f(\vec{\delta}\,\vec{\delta}\,\mathbf{3}) \\ \frac{d^2}{dx^2} G(3) &= 6g(\vec{\delta}\,\vec{\delta}\,\mathbf{3}). \end{aligned}$$

Now, the weights of $\vec{\delta}\,\vec{\delta}\,\mathbf{3}$ for f are

$$\begin{aligned} \vec{\delta}\,\vec{\delta}\,\mathbf{3} &= (-\mathbf{2} + \mathbf{3})\,\vec{\delta}\,\mathbf{3} = -\mathbf{2}\,\vec{\delta}\,\mathbf{3} + \mathbf{3}\,\vec{\delta}\,\mathbf{3} \\ &= \frac{1}{2}\mathbf{1\,2\,3} - \mathbf{2\,3\,3} + \frac{1}{2}\mathbf{3\,4\,3} \\ &= 0\,\mathbf{0\,1\,2} + \frac{1}{6}\mathbf{1\,2\,3} - \frac{7}{24}\mathbf{2\,3\,4} + \frac{1}{8}\mathbf{4\,6}. \end{aligned}$$

Similarly, the weights of $\vec{\delta}\,\vec{\delta}\,\mathbf{3}$ for g are

$$\begin{aligned} \vec{\delta}\,\vec{\delta}\,\mathbf{3} &= -\mathbf{3}\,\vec{\delta}\,\mathbf{3} + \mathbf{4}\,\vec{\delta}\,\mathbf{3} \\ &= \frac{1}{2}\mathbf{1\,2\,3} - \frac{5}{6}\mathbf{2\,3\,3} + \frac{1}{3}\mathbf{3\,4\,3} \\ &= \frac{1}{6}\mathbf{1\,2\,3} - \frac{7}{24}\mathbf{1\,2\,3} + \frac{1}{8}\mathbf{2\,3\,4} + 0\,\mathbf{3\,4\,6}. \end{aligned}$$

Thus,

$$\begin{aligned}\frac{d^2}{dx^2}F(3) = 6f(\vec{\delta}\,\vec{\delta}\,\mathbf{3}) &= 1f(\mathbf{1\,2\,3}) - 1.75f(\mathbf{2\,3\,4}) + .75f(\mathbf{3\,4\,6}) \\ &= 1g(\mathbf{1\,2\,3}) - 1.75g(\mathbf{2\,3\,4}) + .75g(\mathbf{3\,4\,6}) \\ &= 6g(\vec{\delta}\,\vec{\delta}\,\mathbf{3}) = \frac{d^2}{dx^2}G(3).\end{aligned}$$

It is no coincidence that F and G meet with C^2, since they are adjacent segments of a B-spline, as discussed in Section 2.6.

Rationale for Linear Blossoms

Although the linear blossom is not as familiar as the multi-affine blossom, it is a very convenient concept for analysis. Because tensors form a linear space, all the concepts and vocabulary of linear algebra become available. These concepts, such as affine combinations, subspaces and bases, simplify definitions and theorems. Thus, this thesis will use linear blossoms for all theoretical discussions, especially in the derivation of the algorithms of Chapter 4. However, the actual implementation of the blossom datatype, discussed in Chapter 3, will only be multi-affine or multi-linear.

The unfamiliarity of linear blossoms need not pose a problem. In actual practice, they are evaluated only at simple tensors composed of only points and they behave exactly the same as the multi-affine blossom. If linear blossoms are evaluated at simple tensors that include vectors, then they behave exactly like multi-linear blossoms. From here on, "blossom" refers to a linear blossom.

Finally, note that "tensor" in this context is different from "tensor-product", as in "tensor-product B-splines".

2.3 Multi-indices and Triangular Arrays

The coefficients of multi-variate polynomials are commonly indexed with multi-indices. A *multi-index* is a tuple of non-negative integers. A multi-index of *dimension* d and *degree* n is written as $\vec{i} = (i_0, \ldots, i_d)$, where $i_0 + \cdots + i_d = n$. Define $|\vec{i}| \equiv i_0 + \cdots + i_d$. Let $\vec{e}_k$ denote the multi-index that has a 1 in the kth spot and 0 everywhere else. The set of all multi-indices of dimension d, degree n is denoted $\mathbb{I}_d^n$. The size of this set is $D(d,n) \equiv \binom{n+d}{d}$, and is equal to the dimension of the space of d-variate, degree n homogeneous polynomials.

A *triangular array of dimension d and degree n* is a set whose elements are indexed by multi-indices: $\{a_{\vec{i}} : \vec{i} \in \mathbb{I}_d^n\}$. Figure 2.6 shows a triangular array of dimension 2 and degree 3.

$$
\begin{array}{cccc}
 & & a_{(3,0,0)} & \\
 & a_{(2,1,0)} & & a_{(2,0,1)} \\
a_{(1,2,0)} & & a_{(1,1,1)} & & a_{(1,0,2)} \\
a_{(0,3,0)} & a_{(0,2,1)} & & a_{(0,1,2)} & & a_{(0,0,3)}
\end{array}
$$

Figure 2.6: Triangular array of dimension 2 and degree 3

2.4 Polynomial Spaces and Bases

For the sake of brevity, the space of scalar-valued homogeneous polynomials will be simply called the *polynomial space*. A basis of the polynomial space is called a *polynomial basis*. A polynomial in a basis is called a *basis polynomial*. The blossom of a basis polynomial is called a *basis blossom*. A basis of the tensor space is a *tensor basis* and an element of the basis is a *basis tensor*.

A polynomial, F, is written as a combination of basis polynomials, $F_{\vec{i}}$, and *coefficients* in the range space, $P_{\vec{i}}$, where $\vec{i} \in \mathbb{I}_d^n$:

$$F(x) = \sum_{\vec{i} \in \mathbb{I}_d^n} F_{\vec{i}}(x) P_{\vec{i}}.$$

Intuitively, the basis polynomials are used as "weighting functions", where $F_{\vec{i}}$ is the "weight" of the point $P_{\vec{i}}$.

The tensor space and the polynomial space are closely related. Theorem 2.2.2 states that the blossom of a polynomial is a linear map from the tensor space. Thus, the blossom of a scalar-valued polynomial is a linear functional from the tensor space. And since there is a one-to-one correspondence between blossoms and polynomials, the polynomial space is the *dual space* of the tensor space.

This duality implies relations between polynomial bases, tensor bases, polynomial coefficients and tensor coordinates. A polynomial basis $\{F_{\vec{i}}\}$ and its dual basis $\{u_{\vec{i}}\}$ is related through the blossom $\{f_{\vec{i}}\}$ of $\{F_{\vec{i}}\}$ by

$$f_{\vec{i}}(u_{\vec{j}}) = \begin{cases} 1, & \text{if } \vec{i} = \vec{j} \\ 0, & \text{if } \vec{i} \neq \vec{j} \end{cases}.$$

Now, let $F(t) = \sum_{\vec{i}} F_{\vec{i}}(t) P_{\vec{i}}$. Then

$$f(u_{\vec{j}}) = \sum_{\vec{i}} f_{\vec{i}}(u_{\vec{j}}) P_{\vec{i}} = P_{\vec{j}}.$$

This equation says that the coefficients of a polynomial correspond to values of its blossom. On the other hand, let $u = \sum_{\vec{i}} \alpha_{\vec{i}} u_{\vec{i}}$ (i.e. $\alpha_{\vec{i}}$ are the coordinates of u). Then

$$f_{\vec{j}}(u) = f_{\vec{j}}(\sum_{\vec{i}} \alpha_{\vec{i}} u_{\vec{i}}) = \sum_{\vec{i}} \alpha_{\vec{i}} f_{\vec{j}}(u_{\vec{i}}) = \alpha_{\vec{j}}.$$

This equation says that the coordinates of a tensor correspond to the values of the basis blossoms.

For example, the above observations can be used to compute the coefficients of F and coordinates of u over the cubic Hermite basis. The definition of the cubic Hermite basis requires that the polynomials satisfy the following equations [11]:

$$\begin{array}{llll}
H_{(3,0)}(0) = 1, & \frac{d}{dx}H_{(3,0)}(0) = 0, & \frac{d}{dx}H_{(3,0)}(1) = 0, & H_{(3,0)}(1) = 0 \\
H_{(2,1)}(0) = 0, & \frac{d}{dx}H_{(2,1)}(0) = 1, & \frac{d}{dx}H_{(2,1)}(1) = 0, & H_{(2,1)}(1) = 0 \\
H_{(1,2)}(0) = 0, & \frac{d}{dx}H_{(1,2)}(0) = 0, & \frac{d}{dx}H_{(1,2)}(1) = 1, & H_{(1,2)}(1) = 0 \\
H_{(0,3)}(0) = 0, & \frac{d}{dx}H_{(0,3)}(0) = 0, & \frac{d}{dx}H_{(0,3)}(1) = 0, & H_{(0,3)}(1) = 1
\end{array}$$

These equations are uniquely satisfied by the following polynomials:

$$\begin{aligned}
H_{(3,0)}(x) &= (1-x)^3 + 3x(1-x)^2 \\
H_{(2,1)}(x) &= x(1-x)^2 \\
H_{(1,2)}(x) &= -x^2(1-x) \\
H_{(0,3)}(x) &= 3x^2(1-x) + x^3.
\end{aligned}$$

Since $F(x) = f(x^3)$ and $\frac{d}{dx}F(x) = 3f(\vec{\delta}x^2)$, it is easy to see that the dual basis tensors are

$$u_{(3,0)} = \mathbf{0\,0\,0}, \quad u_{(2,1)} = \frac{1}{3}\vec{\delta}\,\mathbf{0\,0}, \quad u_{(1,2)} = \frac{1}{3}\vec{\delta}\,\mathbf{1\,1}, \quad u_{(0,3)} = \mathbf{1\,1\,1}.$$

In this case, all the tensors are simple. The coefficients of F are obtained by computing

$$f(\mathbf{0\,0\,0}), \quad f(\frac{1}{3}\vec{\delta}\,\mathbf{0\,0}), \quad f(\frac{1}{3}\vec{\delta}\,\mathbf{1\,1}), \quad f(\mathbf{1\,1\,1}).$$

On the other hand, the coordinates of u over the basis $u_{(3,0)}$, $u_{(2,1)}$, $u_{(1,2)}$, $u_{(0,3)}$ can be computed by evaluating $h_{(3,0)}(u), \ldots, h_{(0,3)}(u)$. For example, the formula for the multi-linear blossom $\hat{h}_{(2,1)}$ is

$$\hat{h}_{(2,1)}(x_1, x_2, x_3) = \frac{1}{3}\left(x_1(1-x_2)(1-x_3) + x_2(1-x_1)(1-x_2) + x_2(1-x_1)(1-x_3)\right).$$

This is the formula for the linear blossom $h_{(2,1)}$ on simple tensors. The value of $h_{(2,1)}$ on all other tensors is computed by taking linear combinations. Thus,

$$h_{(2,1)}(\mathbf{3\,6\,5} - \mathbf{2\,1\,1}) = \hat{h}_{(2,1)}(3,6,5) - \hat{h}_{(2,1)}(2,1,1) = \frac{158}{3} - 0 = \frac{158}{3}$$

2.5 B-bases

By now it is obvious that the tensors **0 1 2**, **1 2 3**, **2 3 4**, and **3 4 6** used in the examples form a basis: there are four of them and their linear combinations span all tensors. They are an example of a *B-basis*, the most important class of polynomial bases for CAGD. B-bases are the basis polynomials for B-patches [17], and are generalizations of segments of B-spline bases and Bézier bases (of any dimension).

B-bases are easier to analyze using the tensor space rather than the polynomial space. As tensors, they have a simple formula, but as polynomials, they do not. Previously, the polynomials have been defined by a recurrence relation. Using the tensor definition, the recurrence relation becomes a theorem (see Section 4.1).

B-basis tensors are defined using a knot net. A *knot net* is an array of $n(d+1)$ domain points $\{t_{k,l} : k = 0..d, l = 0..n-1\}$, where d is the dimension of the domain and n is the degree of the polynomial space. The points $t_{k,l}$ are called *knots*. The array looks like this:

$$\begin{pmatrix} t_{0,0} & \cdots & t_{0,n-1} \\ \vdots & \ddots & \vdots \\ t_{d,0} & \cdots & t_{d,n-1} \end{pmatrix}.$$

The $\vec{i}$*th evaluation basis* of the knot net is the set of $d+1$ domain points $\{t_{0,i_0}, t_{1,i_1}, \ldots, t_{d,i_d}\}$, consisting of one point selected from each row of the array. These sets must form a basis of the domain space for all $|\vec{i}| \le n-1$; otherwise the knot net does not define a valid B-basis.

Define

$$t_{\vec{i}} \equiv \prod_{k=0}^{d} \prod_{l=0}^{i_k-1} t_{k,l} = t_{0,0} \cdots t_{0,i_0-1} \cdots t_{d,0} \cdots t_{d,i_d-1}.$$

For example, $t_{(2,0,1,1)} = t_{0,0}\, t_{0,1}\, t_{2,0}\, t_{3,0}$, and $t_{\vec{i}}\, t_{k,i_k} = t_{\vec{i}+\vec{e}_k}$. The *B-basis* over the knot net $\{t_{k,l}\}$ consists of the tensors $\{t_{\vec{i}} : |\vec{i}| = n\}$. In other words, the $\vec{i}$*th basis tensor* of the basis is $t_{\vec{i}}$. Note that for all $m \le n$, the subarray of the knot net, $\{t_{k,l} : k = 0..d, l = 0..m-1\}$, forms a basis for the space of m-tensors.

An example of a B-basis is the following. The knot net

$$\begin{pmatrix} \mathbf{2} & \mathbf{1} & \mathbf{0} \\ \mathbf{3} & \mathbf{4} & \mathbf{6} \end{pmatrix}$$

produces the basis tensors

$$\begin{aligned} t_{(3,0)} &= t_{0,0}\, t_{0,1}\, t_{0,2} = \mathbf{2\,1\,0} \\ t_{(2,1)} &= t_{0,0}\, t_{0,1}\, t_{1,0} = \mathbf{2\,1\,3} \\ t_{(1,2)} &= t_{0,0}\, t_{1,0}\, t_{1,1} = \mathbf{2\,3\,4} \\ t_{(0,3)} &= t_{1,0}\, t_{1,1}\, t_{1,2} = \mathbf{3\,4\,5}. \end{aligned}$$

Note that this knot net satisfies the condition that the $\vec{i}$th evaluation basis forms a basis for all $|\vec{i}| < 3$. For example, the $(1,1)$th evaluation basis is $\{t_{(0,1)}, t_{(1,1)}\} = \{\mathbf{1}, \mathbf{4}\}$. In the one-dimensional case, the condition implies that $t_{0,k} \neq t_{1,l}$ for $k + l < 3$. A sufficient condition is that $t_{0,k} < t_{1,l}$ for all k, l. This sufficient condition is satisfied by the example and by B-splines, as discussed in the Section 2.6.

In the two-dimensional example of Figure 2.3, the knot net is

$$\begin{pmatrix} a_0 & a_1 \\ b_0 & b_1 \\ c_0 & c_1 \end{pmatrix}.$$

In this case, the condition on evaluation bases requires that the points in the sets $\{a_0, b_0, c_1\}$, $\{a_0, b_1, c_0\}$, $\{a_1, b_0, c_0\}$, and $\{a_0, b_0, c_0\}$ are not collinear. A sufficient condition for the two-dimensional case is that the knots $\{a_k\}$, $\{b_k\}$, and $\{c_k\}$ are contained in disjoint circles, as in Figure 2.3.

2.5.1 Bézier and Monomial Bases

Bézier bases are a special case of B-bases where knots $t_{k,0} = \cdots = t_{k,n} = t_k$. Thus the elements of the Bézier basis are $\{t_{\vec{i}} = t_0^{i_0} \cdots t_d^{i_d} : |\vec{i}| = n\}$. In the usual definition of Bézier bases, the knots t_k are required to be points, but it is convenient to remove this restriction for this thesis. In this thesis, all such bases are regarded as Bézier bases.

The *monomial* (or power) basis $(1, x, x^2, \ldots, x^n)$ is closely related to the Bézier basis where the t_k are taken from a domain frame, $\{x_0, \vec{v}_1, \ldots, \vec{v}_d\}$. The dual basis tensors of the monomial basis are $\{t_{\vec{i}} = \binom{n}{\vec{i}} x_0^{i_0} \vec{v}_1{}^{i_1} \cdots \vec{v}_d{}^{i_d} : |\vec{i}| = n\}$, where $\binom{n}{\vec{i}}$ is the multinomial coefficient. The coefficients of polynomials over the monomial basis are

$$P_{\vec{i}} = \binom{n}{\vec{i}} f(x_0^{i_0} \vec{v}_1{}^{i_1} \cdots \vec{v}_d{}^{i_d}) = \binom{n}{\vec{i}} n(n-1) \cdots (n - |\vec{i}| + i_0 + 1) \frac{d}{d\vec{v}_1{}^{i_1} \cdots \vec{v}_d{}^{i_d}} F(x_0).$$

This equation means that every coefficient is a vector except one, $f(x_0^n)$.

2.6 Blossoming B-splines

In one dimension, B-bases are segments of B-spline bases. The relationship between B-splines and blossoms was discovered independently by Ramshaw and de Casteljau [16, 7].

Let F be a B-spline curve with knot sequence $t_0, \ldots, t_{n+2L-1}$ and control vertices $P_0, \ldots, P_{n+L}$. Let F_i be the segment of the B-spline over the interval $[t_{i+n-1}, t_{i+n})$, where $t_{i+n-1} < t_{i+n}$ and $0 \leq i \leq L-1$. Then the blossom f_i of F_i has the blossom values

$$f_i(t_i \cdots t_{i+n-1}) = P_i, \quad \ldots, \quad f_i(t_{i+n} \cdots t_{i+2n-1}) = P_{i+n}. \tag{2.2}$$

Thus, $P_i, \ldots, P_{i+n}$ correspond to the coefficients of f_i over the knot net

$$\begin{pmatrix} t_{0,0} = t_{i+n-1} & \cdots & t_{0,n-1} = t_i \\ t_{1,0} = t_{i+n} & \cdots & t_{1,n-1} = t_{i+2n-1} \end{pmatrix}.$$

For example, given a B-spline with knot sequence $0, 0, 0, 1, 2, 3, 4, 6, 6, 6$ and control points $P_0, P_1, P_2, P_3, P_4, P_5, P_6$, the segment in the interval $[2, 3)$ is a blossom that has knot net

$$\begin{pmatrix} \mathbf{2} & \mathbf{1} & \mathbf{0} \\ \mathbf{3} & \mathbf{4} & \mathbf{6} \end{pmatrix},$$

and coefficients P_2, P_3, P_4, P_5 (Figure 2.7).

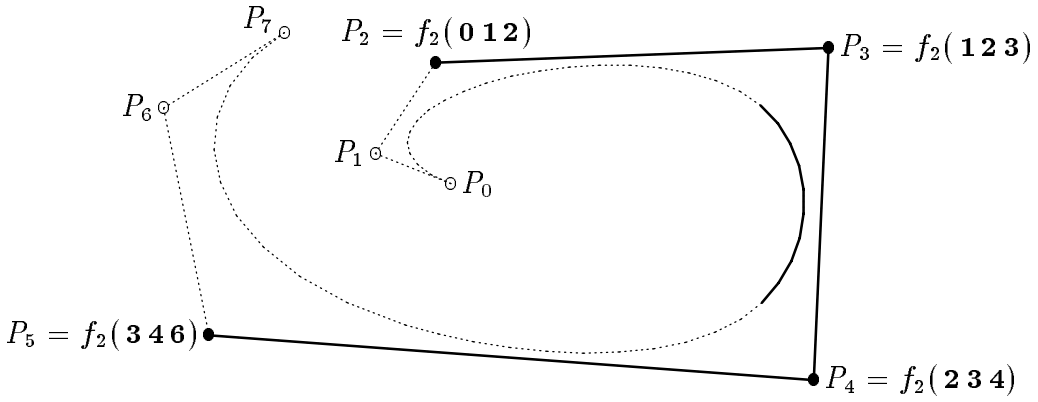


Figure 2.7: Blossom of a segment of a B-spline

Chapter 3

Datatype for Blossoming

This thesis aims to simplify the programming of curve and surface modeling paradigms. It can aid research into new modeling paradigms by allowing researchers to implement them more easily. The basic approach is to create a system that provides blossoming datatypes.

In this chapter, I motivate the use of blossoming datatypes. Then I give an overview of previous work using this approach, and analyze their strengths and weaknesses. Finally I give an overview of the new system that uses this approach, describing its design criteria, and explaining how the design meets the criteria.

3.1 Rationale for Blossom Datatypes

To implement a modeling paradigm, one translates the mathematical derivation into code that performs the required computations. The mathematical description for many piecewise polynomial paradigms uses blossoming analysis.

The benefit of blossoming analysis is that it is simple yet powerful. It does not require complex mathematics; it has one simple concept, the blossom. It is also geometrically intuitive. It is powerful because it unifies many popular paradigms. Under the blossoming approach, different paradigms are distinguished by three properties: the tensor basis, the domain, and the range. Curves are obtained by one-dimensional domains, surfaces by two-dimensional. B-splines, Bézier curves and surfaces are some examples of paradigms obtained by different tensor bases. Rational polynomial paradigms are obtained by projective range spaces.

Despite its simplicity, the blossoming description is difficult to translate to computer code. The problem is that blossoming concepts must be translated to computer language concepts, such as floating-point arithmetic. However, the datatype concept solves this difficulty. A datatype is simply an abstract set of objects with operations that can be performed on the objects. In this case, the objects correspond to concepts used in the blossoming analysis, such as blossoms, tensors, bases, spaces or multi-indices. The operations perform meaningful

actions on the objects in terms of blossoming analysis. Thus, the programmer can use the operations to manipulate the mathematical concepts in the code.

Blossoming datatypes facilitate research by making modeling prototypes easier to write. The datatypes also make programs easier to read. They make it easy to see whether a program is correct by seeing whether the operations manipulate the concepts correctly. Thus, programs become easier to maintain and change.

Blossoming concepts can be made into datatypes very naturally. First, blossoming gives a unified representation of different paradigms: the representation of a curve or surface is given by a set of basis tensors and a set of coefficients. How these two sets of parameters are assigned depends on the application. For example, they may be set interactively or as a result of filtering data.

Second, blossoming gives a unified view of the operations on different paradigms. Operations extract various kinds of information from the representation. For example, an application may want to obtain the location of various points on a surface, or derivatives, or a bounding box. Since a blossom is conceptually a function, obtaining any kind of information corresponds to evaluating the blossom at certain tensors.

3.2 Previous Work

DeRose and Goldman [9] first proposed the approach of using blossoming datatypes. Their proposed system extends the coordinate-free geometry programming package [8] with a blossom datatype. Their datatype supports the two fundamental operations of creation and evaluation. Blossoms are defined by their coefficients over Bézier bases, and are evaluated using an extension of the de Casteljau algorithm.

While DeRose and Goldman's proposed system was not implemented, it would have made it straight-forward to translate a blossoming analysis to an implementation: simply use blossom evaluation to compute the desired blossom values. However, many algorithm cannot be efficiently implemented with only evaluation. Also, being limited to Bézier bases prevented the system from implementing many paradigms.

The major work implemented using the blossoming datatype approach was by Dahl. He provided a blossom datatype in *Weyl* [5], a language for CAGD research. The Weyl language provides datatypes that closely mimics the corresponding mathematical concepts. Weyl also has an interactive, graphical environment. The goal of Weyl was to provide an environment where researchers can manipulate mathematical objects in familiar ways and receive graphical feedback.

In Weyl, a blossom is conceptually a function. Operations are provided that mimic mathematical operations on function. For example, "high-level" operations like affine combination, dot product, degree-raise and composition are provided. The basic operations of blossom evaluation and partial evaluation can be used to create new algorithms. Partial evaluation allows more efficient algorithms to be created than with evaluation alone.

A blossom is defined by supplying coefficients and an arbitrary basis of simple tensors. For convenience, special functions are provided to generate the bases for Lagrange, B-bases, Bézier and power bases. Internally, however, all blossoms are stored over a standard Bézier basis.

The Weyl system is sufficient for its intended purposes, but is not suitable for general research use. Its biggest drawback is that the datatypes are tied to the Weyl language environment; they cannot be used in another environment, or in conjunction with other tools. This is a serious problem as the Weyl environment does not provide all the necessary facilities that researchers need. For example, if a surface fitting scheme requires the use of singular value decomposition, the function would have to be created in the Weyl language. It is much more convenient to use one of the many existing packages for matrix manipulation available in C or Fortran libraries.

Second, the system emphasizes mathematical purity but is not concerned with efficiency. For example, there is no efficient way to get coefficients over a basis, or move a control point. Also, the system may encounter numerical stability problems because it converts all blossoms to a standard Bézier basis.

Finally, Weyl is a functional programming language (based on Scheme). The functional style of programming is seldom used in CAGD, and is less familiar to researchers, who usually use imperative languages like Fortran or C.

3.3 New System: The Blossom Classes

In this section, I present the Blossom Classes, a new system to support programming with blossoming datatypes. First, I give the requirements for the system, and after that, I present an overview of the system, and discuss the design decisions and trade-offs.

3.3.1 Requirements for System

A major goal of the Blossom Classes is to be generally useful, meaning it can support many different applications of blossoms. This goal implies, firstly, that the system should handle the most general case possible. Secondly, the system's functionality should be, in some sense, complete. Since completeness is difficult to achieve, the system should provide basic building blocks that can be used to create more functionality. The system should also be able to work with other tools to increase functionality. Thirdly, the system should be efficient, numerically stable, and easy to use.

Some of these design criteria conflict with each other. Having to handle the general case means special cases are not handled as efficiently. Also, the general case is more complex than special cases, making the system more difficult to use. To resolve this conflict, the system should allow the user to customize parts of the system for special cases.

These criteria hold implications for both the design of the datatype (what object and

operations should be provided), and how the datatype should be packaged (how to make it easy for programmers to use the datatype in actual programs).

A good datatype has operations that are easy to use, and perform meaningful computations. The computations should be performed by efficient and numerically stable algorithms. A set of basic building-block operations should be provided, from which the user can derive new computations easily. Finally, the operations should support the familiar procedural style of programming.

To be widely used, the datatype should be packaged as a library that users can include into their own systems. The library should be able to inter-operate with many systems and tools. For example, it should work with data structures and routines created by the user or taken from other libraries. Further, in special situations where an object or operation can be implemented more efficiently, the user should be able to use a specialized implementation with the rest of the library.

3.3.2 Overview of Design

I wrote the Blossom Classes as a C++ library for several reasons: C++ is a popular language for CAGD applications, and is familiar to programmers. C++ works well with many system and tools, and even other popular languages like Fortran or C.

Because it is a library, users can easily incorporate the Blossom Classes into their applications, and can use it with other tools, like matrix libraries. The Blossom Classes library is part of the Computer Graphics Lab Splines project [2] at the University of Waterloo, and works with the datatypes in the Splines library. In addition, the Blossom Classes library is specifically designed to work with the Standard Template Library (STL) [21], which is a library of generic algorithms designed to work with many different classes. STL is part of the new C++ standard, and thus will be available to all C++ users.

The outstanding feature of the Blossom Classes library is that it works with user-supplied classes. Like STL, it is designed so that a datatype can have many implementations by different classes. The idea is to specify a datatype by an abstract interface, i.e., as a set of functions that a class must provide in order to implement that datatype. The library code manipulates the datatype using only the functions defined in the abstract interface. Thus, users can integrate their own classes with the library simply by providing the functions required in the abstract interface. Users can also use this facility to create specialized implementations that are more efficient for certain applications.

The library has three components: the blossom datatypes, the geometric datatypes, and the operations on blossoms. The heart of the library is the blossom operations. These operations operate on the blossoming datatypes through abstract interfaces. The blossoming datatypes, in turn, use the geometry datatypes through abstract interfaces.

3.3.3 Datatypes

Blossoming Datatypes

The design of the blossom datatype has several important features. Firstly, I decided to support only B-bases in the library. This decision means other useful bases, like Lagrange bases, are not directly supported. I made this choice because B-bases have the only known efficient and numerically stable algorithms to evaluate blossoms and perform other useful operations (see Section 4). The only way to handle arbitrary bases is to implicitly convert all blossoms to B-bases to do operations; this is inefficient and may be numerical unstable. The user can still choose to convert bases explicitly, so the library supports other bases indirectly.

On the other hand, arbitrary B-bases are supported. Thus, many useful paradigms can be implemented: Bézier curves or surfaces, B-splines, B-patches, monomials. This also means blossoms are not converted to a standard basis, avoiding numerical instability problems.

Secondly, multi-affine or multi-linear blossoms are directly supported, but not affine or linear blossoms: blossoms can be directly evaluated at simple tensors, not arbitrary tensors. Again, this decision was made because of what algorithms are available. This decision does not limit the usefulness of the library since most blossoming analysis uses the multi-affine version. These blossoms are also more familiar to people and are easier to work with. Users can still evaluate at arbitrary tensors by converting to a standard basis.

Thirdly, I decided to view the blossom as an array of coefficient over a knot net, rather than following the Weyl idea of a blossom as a function. In a procedural language, it is more natural to work with blossoms by setting the knots and the coefficients directly.

As a result of this view of blossoms, the library defines datatypes for triangular arrays, multi-indices and knot nets. Triangular array datatypes are required to satisfy STL *container* datatype requirements. The library actually provides two datatypes for knot nets, one for arbitrary B-bases and one for the special case of Bézier bases. The Bézier datatype is provided because it is more efficient for some operations. However, Bézier knot nets do not support certain other operations.

The library provides classes that implement these datatypes. Class `Blossom<domain,range>` is a templated blossom class that works with any kind of domain and range classes. It uses the classes `TriArray`, `MultiIndex`, and `KnotNet` which provide triangular arrays, multi-indices, and knot nets of arbitrary dimension and degree. Class `TriArray<T>` can store objects of any type `T`. Class `MultiIndex` stores a variable-length array of integers. Class `KnotNet<domain>` stores a two-dimensional array of knots of any domain type. Class `BezKnotNet<domain>` implements a Bézier knot net and stores a domain basis. No blossom class is provided for `BezKnotNet`.

User-supplied types can be used instead of any of these classes (so long as certain functions and operators are provided). This is useful, for example, when the application only deals with curves. In that case, more efficient implementations of these classes can be used.

Geometry Datatypes

The library provides datatypes for working with domain and range spaces and scalars. I made these datatypes separate from the blossoming datatypes that use them. This separation provides transparency of operation: the actual implementation of geometric datatypes can change without any change in the blossoming datatypes.

The flexibility of using different types for the geometry component means many different paradigms can be obtained: one-dimensional domains for curves, two-dimensional for surfaces; affine domains for polynomials, linear domain for homogeneous polynomials; projective range for rational polynomials. Even more exotic spaces can be used. For example, Shoemake used Bézier curves that map to the unit quaternions to control rotation [19]. Finally, while real numbers are usually used for scalars, it is possible to use complex numbers.

The library provides the `PtDomain` class, which implements linearized spaces of arbitrary dimension. The class can also be used as a projective range space. The library also supports the built-in type `double` as an efficient one-dimensional affine space. Both classes use `double` as the scalar type.

3.3.4 Blossom Operations

The blossom operations operate on blossoming datatypes through their abstract interfaces. Thus, these operations can work with different classes. The operations are useful basic building-block operations and can be used to implement different algorithms (see Chapter 6). The operations are based on efficient algorithms for B-bases, discussed in Section 4.

Defining Blossoms

As discussed in Section 3.1, to get a representation of a curve or surface means assigning the basis tensors and the coefficients. The blossom datatype provides functions to set the knots and coefficients.

The operation

```
f.setCoeff(i,P);
```

will set the `i`th coefficient of the blossom `f` to the range point `P`, where `i` is a multi-index object. The operation

```
f.setKnot(k,l,x);
```

sets the `(k,l)` knot of `f` to the domain point `x`, where `k` and `l` are `int`s. The operation

```
f.setDomainElement(k,x);
```

sets all the knots `(k,0)`,`(k,1)`, ... to `x`.

$f.coeffs = \{P_0, P_1, P_2, P_3\}$ $\quad \vec{i} = (2,1)$ `f.setKnot(1,0,4)`

$f.knots = \begin{pmatrix} \mathbf{2} & \mathbf{1} & \mathbf{0} \\ \mathbf{3} & \mathbf{4} & \mathbf{6} \end{pmatrix}$ `f.setCoeff(i,P)`

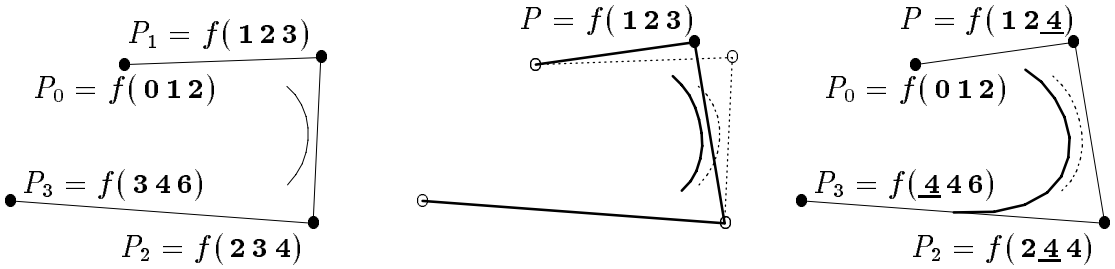


Figure 3.1: Operations for defining blossoms

Bézier knot nets do not support the `setKnot()` operation, but do support `setDomainElement()`.

Figure 3.1 shows the effect of these operations on a curve segment. By convention, the segment is defined to be over the interval $[t_{0,0}, t_{1,0})$. This convention reflects the relationship to B-splines (Section 2.6). Thus, in the left two pictures, the segment is over $[2,3)$, but in the right-hand picture, the segment is over $[2,4)$.

Evaluating Blossoms

Evaluating the blossom extracts information from the representation. Partial evaluation is also provided for reusing intermediate results of an evaluation.

The operations

```
P = eval(f,args);
P = eval(kn,a,args);
```

will evaluate a blossom at the list of range points stored in `args` and return a point in the range. The first version evaluates the blossom object `f`, while the second version evaluates the blossom implicitly defined by the array of coefficients `a`, over the knot net `kn`. The `args` parameter is any STL *forward iterator* object that returns points in the domain. Thus, the library takes advantage of datatypes already defined by STL instead of defining new ones.

The operations

```
P = diagonalEval(f,x);
P = diagonalEval(kn,a,x);
```

evaluate a blossom at the same argument, `x`, n times.

The operations

```
f2 = partialEval(f,argsbegin,argsend,(blossom*)0);
partialEval(kn,a,a2,args);
```

$args = \{2.7, 3.4, 2.2\}$

```
P = eval(
 f,args.begin());
```

```
f2 = partialEval(
 f,2.7,(blossom*)0);
```

```
f3 = partialEval(
 f2,3.4,(blossom*)0);
```

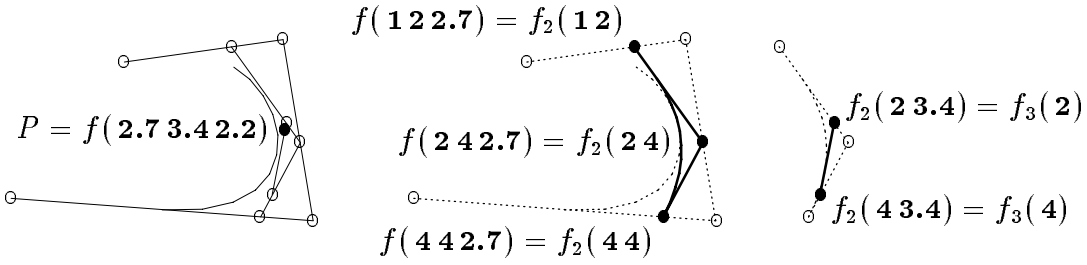


Figure 3.2: Operations for evaluating blossoms

returns a new blossom that is the partial evaluation of a blossom. The first version takes a blossom object `f` and returns an object of type `blossom`, where the type is indicated by the `(blossom*)0` parameter. The return type can be different from the type of `f`. The `argsbegin` parameter is an iterator that marks the beginning of the list of arguments, while `argsend` marks the end. The second version puts the coefficients of the partial evaluation into the array `a2`. This version doesn't need to know `argsend` because the size of `a2` tells it how many arguments it must evaluate. Also `a` and `a2` can be different types.

As noted in Section 3.3.3, these operations evaluate blossoms at simple tensors only. Also, Bézier bases compute these operations more efficiently than arbitrary B-bases.

Figure 3.2 shows the evaluation of a one-dimensional blossom.

Swapping Knots

Finally, knot swapping operations are provided. These operations compute the coefficients of a blossom over an altered knot net. They are also useful because they perform their calculations "in place", and do not need to allocate extra memory, unlike the evaluation operations.

Note that while these operations change both the coefficients and the knots of the blossom, they do not change the *function that the blossom represents*. That is, evaluating a blossom at the same arguments before and after swapping will yield the same results.

The operations

```
knotReplaceCoeffs(f,k,x);
knotReplaceCoeffs(kn,a,k,x);
```

compute the coefficients of the blossom over a new knot net that has `x` at position `(k,n-1)`, where `n` is the degree of the blossom.

The operations

```
knotSwapCoeffs(f,k,from,to);
knotSwapCoeffs(kn,a,k,from,to);
```

compute the coefficients of the blossom over the new knot net, where

1. the knot at `(k,from)` is moved to position `(k,to)`, and
2. if `from<to`, the knots `(k,from+1)..(k,to)` are moved forward one position to `(k,from)..(k,to-1)`, otherwise, they are moved backward one position.

The operations

```
knotSwapCoeffs(f,k,from,to,num);
knotSwapCoeffs(kn,a,k,from,to,num);
```

swap multiple knots efficiently. The first version is equivalent to

```
knotSwapCoeffs(f,k,from,to);
knotSwapCoeffs(f,k,from+1,to+1);
...
knotSwapCoeffs(f,k,from+num-1,to+num-1);
```

The user must be careful in using these operations as they may cause a knot net to become invalid (i.e. some evaluation basis may become degenerate).

Bézier knot nets do not support these operations.

Figure 3.3 shows the effect of these operations on a curve segment. Note that in the left-hand picture, the knot net corresponds to a B-spline with knot sequence (0,1,2,4,4,2.7). This sequence is not legal since B-spline knot sequences must be in increasing order. One result of this violation is that the curve segment no longer fits inside the convex hull of the control points. The middle picture corresponds to the legal knot sequence (0,1,2,2.7,4,4). Note that the two operations of `knotReplaceCoeffs` and `knotSwapCoeffs` has effectively computed a knot insertion of the B-spline. The right-hand picture shows the reverse operation, knot deletion.

In the one-dimensional case, the two operations `knotReplaceCoeffs` and `knotSwapCoeffs` are usually used together because of the relationship to B-splines. However, these operations extend to higher dimensions, and in that case, there may be occasion to use them separately.

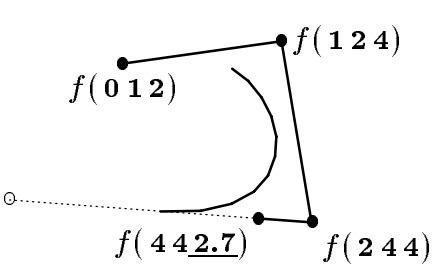
Tensor Operations

The following operations are provided to manipulate tensors. They manipulate the array of coordinates of the tensor over a knot net. The operation

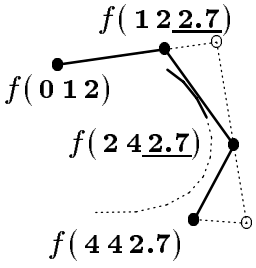
```
getCoords(a,kn,args);
```

computes the coordinates over knot net `kn` of the simple tensor stored in `args`. It places the coordinates in `a`, which is a triangular array storing scalars. The operation

```
knotReplaceCoeffs(
 f,1,2.7);
```




```
knotSwapCoeffs(
 f,1,2,0);
```




```
knotSwapCoeffs(f,1,0,2);
knotReplaceCoeffs(f,1,6.0);
```

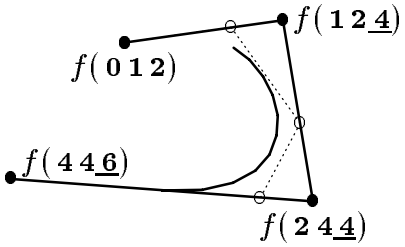


Figure 3.3: Operations for swapping knots of a blossom

```
getPartialCoords(a,a2,kn,args);
```

takes a tensor whose coordinates are stored in `a` and multiplies that tensor by the simple tensor stored in `args`, and places the resulting coordinates in `a2`. The operations

```
knotSwapCoords(a,kn,from,to);
knotSwapCoords(a,kn,from,to,num);
knotReplaceCoords(kn,a,k,x);
```

compute the new coordinates of the tensor over a modified knot net. See the corresponding operations `knotSwapCoeffs` and `knotReplaceCoeffs` for the effect on the knot net. Bézier knot nets do not support these operations.

Other operations on tensors, like computing combinations of tensors, can be performed using STL algorithms on the triangular arrays of coordinates. Thus, the library does not provide these operations.

Chapter 4

Algorithms for B-bases

One reason B-splines and Bézier bases are so important in CAGD is that they have efficient and numerically stable algorithms that perform useful computations. Work has been done to extend these algorithms to multi-affine blossoms of curves and surfaces [9, 1, 12]. In creating the operations for the blossom datatype, I discovered these algorithms naturally generalize to operations on arbitrary B-bases. All the algorithms are based on a fundamental recurrence relation that B-bases satisfy. In this chapter I derive several algorithms using this recurrence.

4.1 Fundamental Recurrence of B-bases

The fundamental recurrence relation of B-basis polynomials has been known in various forms, and has been used as the definition of B-bases (see for example Lodha and Goldman's paper [14]). This section generalizes the recurrence to B-basis tensors.

Given a knot net $\{t_{k,l} : k = 0..d, l = 0..n-1\}$, $\vec{\imath} \in \mathbb{I}_d^{n-1}$, and a domain point x, there exists a relation between $t_{\vec{\imath}}x$ and $\{t_{\vec{\imath}+\vec{e}_k} : k = 0..d\}$. Let $\lambda_k^{\vec{\imath}}$ be the coordinates of x relative to the $\vec{\imath}$th evaluation basis of the knot net. Then,

$$t_{\vec{\imath}}\, x = t_{\vec{\imath}} \left(\sum_{k=0}^{d} \lambda_k^{\vec{\imath}} t_{k,i_k} \right) = \sum_{k=0}^{d} \lambda_k^{\vec{\imath}} t_{k,i_k} t_{\vec{\imath}} = \sum_{k=0}^{d} \lambda_k^{\vec{\imath}} t_{\vec{\imath}+\vec{e}_k} \tag{4.1}$$

This recurrence is known as the *up-recurrence*.

The recurrence can be reversed, to get a relation between the dual basis polynomials.

Let $f_{\vec{i}}$ be the dual basis blossoms of $t_{\vec{i}}$.

$$\begin{aligned} f_{\vec{i}}(xu) &= f_{\vec{i}}(x \sum_{\vec{j} \in \mathbb{I}_d^{n-1}} f_{\vec{j}}(u) t_{\vec{j}}) \\ &= \sum_{\vec{j} \in \mathbb{I}_d^{n-1}} f_{\vec{j}}(u) f_{\vec{i}}(x t_{\vec{j}}) \\ &= \sum_{\vec{j} \in \mathbb{I}_d^{n-1}} f_{\vec{j}}(u) f_{\vec{i}}(\sum_{k=0}^{d} \lambda_k^{\vec{j}} t_{\vec{j}+\vec{e}_k}) \\ &= \sum_{\vec{j} \in \mathbb{I}_d^{n-1}} \sum_{k=0}^{d} \lambda_k^{\vec{j}} f_{\vec{j}}(u) f_{\vec{i}}(t_{\vec{j}+\vec{e}_k}) \\ f_{\vec{i}}(xu) &= \sum_{k=0}^{d} \lambda_k^{\vec{j}} f_{\vec{i}-\vec{e}_k}(u) \end{aligned} \tag{4.2}$$

This recurrence is a generalization of the Cox–de Boor–Mansfield recurrence for B-splines, and is known as the *down recurrence*.

4.2 Algorithms on Coefficients and Coordinates

Equation 4.1 can be used to compute a new blossom value $f(xt_{\vec{i}})$ from given blossom values $f(t_{\vec{i}+\vec{e}_k})$:

$$f(t_{\vec{i}}x) = \sum_{k=0}^{d} \lambda_k^{\vec{i}} f(t_{\vec{i}+\vec{e}_k}).$$

This relation leads to the following algorithm:

> The *combine-coefficients* algorithm takes an *evaluation point* x and a multi-index $\vec{i} \in \mathbb{I}_d^{n-2}$. The computation proceeds by first computing the coordinates, $\lambda_k^{\vec{i}}$, of x over the $\vec{i}$th evaluation basis, and then returning the combination $\sum_{k=0}^{d} \lambda_k^{\vec{i}} f(t_{\vec{i}+\vec{e}_k})$.

The de Casteljau algorithm works by performing combine-coefficients repeatedly.

Since the coefficients of f are $P_{\vec{i}} = f(t_{\vec{i}})$ and the coefficients of $f|_x$ are $Q_{\vec{i}} = f(xt_{\vec{i}})$, this relation can be used to compute the coefficients of $f|_x$ from coefficients of f. The set $\{P_{\vec{i}+\vec{e}_k} : k = 0..d\}$ is called the *$\vec{i}$th evaluation subarray* of the coefficient array. Thus, the combine-coefficients algorithm takes the combination of the $\vec{i}$th evaluation subarray with the coordinates of x.

The left side of Figure 4.1 shows the combine-coefficients algorithm in two-dimensions. In the figure, the $(1,0,1)$th evaluation subarray is being combined to compute $f(xt_{(1,0,1)})$. An arrow indicates that the algorithm multiplies the value at the tail of the arrow by the

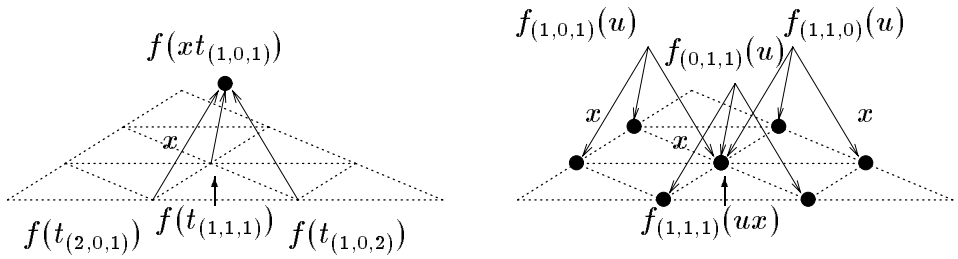


Figure 4.1: Combine-coefficients and weigh-coordinates algorithms.

coordinate of x over the evaluation array ($\lambda_k^{(1,0,1)}$) and adds the result to the value at the head of the arrow.

Similarly, Equation 4.2 can be used to compute $f_{\vec{i}}(ux)$ from given values of $f_{\vec{i}-\vec{e}_k}(u)$. The equation implies that the "contribution" of $f_{\vec{i}-\vec{e}_k}(u)$ to $f_{\vec{i}}(ux)$ is $f_{\vec{i}-\vec{e}_k}(u)\lambda_k^{\vec{i}}$. This relation leads to the following algorithm:

> The *weigh-coordinates* algorithm takes an evaluation point x and a multi-index $\vec{i} \in \mathbb{I}_d^{n-2}$ and a scalar α. The computation proceeds by first getting the coordinates of x over the $\vec{i}$th evaluation basis: $\lambda_k^{\vec{i}}$. Then the algorithm adds the contribution of α to $f_{\vec{i}+\vec{e}_k}(u)$:
>
> $$f_{\vec{i}+\vec{e}_k}(u) \longleftarrow f_{\vec{i}+\vec{e}_k} + \alpha\lambda_k^{\vec{i}}$$

Since the coordinates of u are $\alpha_{\vec{i}} = f_{\vec{i}}(u)$, this relation can be used to compute the $\vec{j}$th coordinate of ux, $\alpha'_{\vec{j}}$ from coordinates of u: start with $\alpha'_{\vec{j}} = 0$ and accumulate the contributions of each $\alpha_{\vec{j}-\vec{e}_k}$, $k = 0..d$. The set $\{\alpha_{\vec{i}+\vec{e}_k} : k = 0..d\}$ is called the *$\vec{i}$th evaluation subarray* of the coordinate array. Thus, the weigh-coordinates algorithm weighs the $\vec{i}$th evaluation subarray by the coordinates of x times α.

The right side of Figure 4.1 shows three weigh-coordinates computations. In the figure, α is set to $f_{(0,1,1)}(u)$, $f_{(1,0,1)}(u)$, and $f_{(1,1,0)}(u)$ in turn. An arrow indicates the algorithm multiplies the value at the tail of the arrow (α) by the coordinate of x over the evaluation array ($\lambda_k^{(1,0,1)}$, $\lambda_k^{(0,1,1)}$ or $\lambda_k^{(1,1,0)}$) and adds the result to the value at the head of the arrow.

4.3 Evaluation and Getting Coordinates

Partial Evaluation and Getting Partial Coordinates

As was noted, the combine-coefficients algorithm can be used to compute the partial evaluation $f|_x$:

> The *partial evaluation* algorithm takes the coefficients of f and runs the combine-coefficients algorithm for all $|\vec{i}| = n - 1$. The resulting points, $f(xt_{\vec{i}})$, are the coefficients of $f|_x$ over the knot net $\{t_{k,l} : k = 0..d, l = 0..n-2\}$.

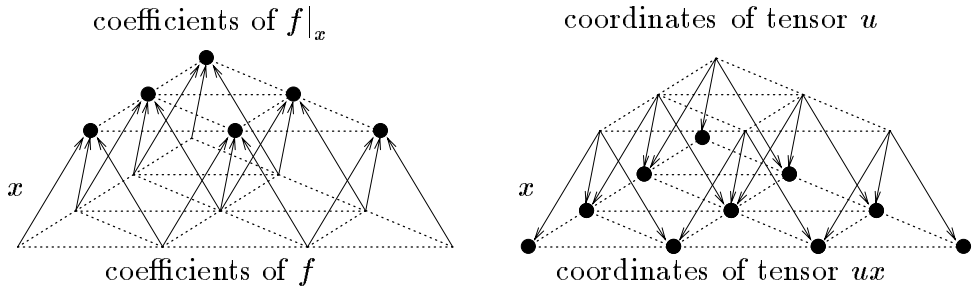


Figure 4.2: Computing partial evaluation and getting partial coordinates.

Similarly, the weigh-coordinates algorithm can be used to compute giving the coordinates of xu from the coordinates of u:

> The *get partial coordinates* algorithm takes the coordinates $\alpha_{\vec{j}}$ of u over $\{t_{k,l} : k = 0..d, l = 0..n-2\}$ and runs weigh-coordinates for all $|\vec{j}| = n-2$. The resulting points $\alpha'_{\vec{i}}$ are the coordinates of xu over $\{t_{k,l} : k = 0..d, l = 0..n-1\}$.

Examples of these algorithms are shown in Figure 4.2.

Full Evaluation and Getting Coordinates

The following algorithm is a generalization of the de Casteljau and de Boor algorithms:

> The *(full) evaluation* algorithm runs partial evaluation successively, to compute the partial evaluations $f|_{x_1}, f|_{x_1 x_2}$, and so on, to $f|_{x_1 \cdots x_n}$. The single coefficient of $f|_{x_1 \cdots x_n}$is the value of $f(x_1 \cdots x_n)$.

Running the de Casteljau algorithm "backwards" yields the algorithm to get the coordinates of a simple tensor:

> The *get (full) coordinates* algorithm runs the get partial coordinates algorithm successively, to compute the the coordinates of x_1 and then $x_1 x_2$ and so on until $x_1 \cdots x_n$.

Examples are shown in Figure 4.3.

Polynomial Evaluation and Basis Polynomial Evaluation

The previous algorithms evaluate arbitrary tensors. To evaluate points on polynomials means to evaluate at the tensor x^n:

> The *diagonal evaluation* algorithm performs a full evaluation at x^n to compute $f(x^n) = F(x)$, thus evaluating a point on the polynomial. On the other hand, the *get diagonal coordinates* algorithm gets the coordinates of x^n to compute $f_{\vec{i}}(x^n) = F_{\vec{i}}(x)$, thus evaluating basis polynomials.

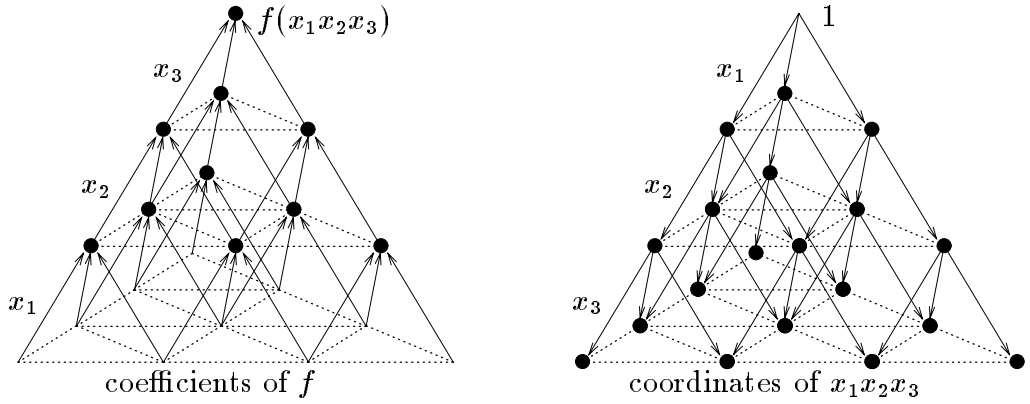


Figure 4.3: Full evaluation and getting coordinates.

Getting diagonal coordinates is an efficient way to evaluate all $D(d,n)$ basis polynomials at x at once.

4.4 Knot Swapping

Knot Swapping

In this section, I derive the relationship between a basis, B, and the basis where two knots of B, $t_{p,q}$ and $t_{p,q+1}$, are swapped. This relationship is used to derive efficient algorithms to compute basis conversions and to insert knots into B-splines (see Chapter 6).

Stated formally, the problem is as follows. Given B with knot net $\{t_{k,l} : k = 0..d, l = 0..n-1\}$, define B' with knot net $\{t'_{k,l} : k = 0..d, l = 0..n-1\}$ where $t'_{p,q} = t_{p,q+1}$, $t'_{p,q+1} = t_{p,q}$ and $t'_{k,l} = t_{k,l}$ for $k \neq p$ or $l \neq q$ or $q+1$. Then, the $\vec{\imath}$th basis tensors of B' is

$$t'_{\vec{\imath}} = \begin{cases} t_{0,0} \cdots t_{p-1,i_{p-1}-1} \underline{t_{p,0} \cdots t_{p,i_p-1}} t_{p+1,0} \cdots t_{d,i_d-1}, & \text{if } i_p < q+1 \\ t_{0,0} \cdots t_{p-1,i_{p-1}-1} \underline{t_{p,0} \cdots t_{p,q+1} t_{p,q} \cdots t_{p,i_p-1}} t_{p+1,0} \cdots t_{d,i_d-1}, & \text{if } i_p > q+1 \\ t_{0,0} \cdots t_{p-1,i_{p-1}-1} \underline{t_{p,0} \cdots t_{p,q-1} t_{p,q+1}} t_{p+1,0} \cdots t_{d,i_d-1}, & \text{if } i_p = q+1 \end{cases}$$

Thus,

$$t'_{\vec{\imath}} = \begin{cases} t_{\vec{\imath}}, & \text{if } i_p < q+1 \\ t_{\vec{\imath}}, & \text{if } i_p > q+1 \\ t_{\vec{\imath}-\vec{e}_p} t_{p,q+1} = \sum_k \lambda_k^{\vec{\imath}} t_{\vec{\imath}-\vec{e}_p+\vec{e}_k}, & \text{if } i_p = q+1 \end{cases} \tag{4.3}$$

where $\lambda_k^{\vec{\imath}}$ are the coordinates of $t_{p,q+1}$ over the $\vec{\imath}$th evaluation basis of B.

Next, I derive the relation between the basis *polynomials* of B and B'. Recall from

Section 2.4 that $u = \sum_{\vec{j}} f'_{\vec{j}}(u)t'_{\vec{j}}$. Now,

$$\begin{aligned} f_{\vec{i}}(u) &= f_{\vec{i}}(\sum_{\vec{j}} f'_{\vec{j}}(u)t'_{\vec{j}}) \\ &= \sum_{\vec{j}} f'_{\vec{j}}(u) f_{\vec{i}}(t'_{\vec{j}}) \\ &= \sum_{j_p \neq q+1} f'_{\vec{j}}(u) f_{\vec{i}}(t'_{\vec{j}}) + \sum_{j_p = q+1} f'_{\vec{j}}(u) f_{\vec{i}}(t'_{\vec{i}}) \\ &= \sum_{j_p \neq q+1} f'_{\vec{j}}(u) f_{\vec{i}}(t_{\vec{j}}) + \sum_{j_p = q+1} \sum_{k} f'_{\vec{j}}(u) \lambda^{\vec{i}}_k f_{\vec{i}}(t_{\vec{i}-\vec{e}_p+\vec{e}_k}). \end{aligned}$$

Expanding the sum and collecting the terms for $t_{\vec{i}}$ yields this relation for $f_{\vec{i}}(u)$:

$$f_{\vec{i}}(u) = \begin{cases} f'_{\vec{i}}(u)\lambda^{\vec{i}}_p, & \text{if } i_p = q+1 \\ f'_{\vec{i}}(u) + \sum_k f'_{\vec{i}+\vec{e}_p-\vec{e}_k}(u)\lambda^{\vec{i}+\vec{e}_p-\vec{e}_k}_k, & \text{if } i_p = q \\ f'_{\vec{i}}(u), & \text{otherwise} \end{cases} \tag{4.4}$$

Note that this formula gives basis polynomials of B in terms of basis polynomials of B', whereas the previous formula (Equation 4.3) gives the basis tensors of B' in terms of the basis tensors of B.

Equation 4.3 can be used to compute coefficients of a blossom, f, over B' from its coefficients over B.

> The *knot swapping* algorithm sets $f(t'_{\vec{i}}) = f(t_{\vec{i}})$ for all $\vec{i}$ such that $i_p \neq q+1$. Then, the algorithm runs the combine-coefficients algorithm over the $(\vec{i}-\vec{e}_p)$th evaluation subarray, for all $\vec{i}$ such that $i_p = q+1$. This computes $f(t'_{\vec{i}}) = \sum_k \lambda^{\vec{i}}_k t_{\vec{i}-\vec{e}_p+\vec{e}_k}$.

Similarly, Equation 4.4 can be used to compute the coordinates of a tensor u over B from its coordinates over B'.

> The *knot swapping for coordinates* algorithm starts with $f_{\vec{i}}(u) = f'_{\vec{i}}(u)$ if $i_p \neq q+1$, and $f_{\vec{i}}(u) = 0$ if $i_p = q$. Then for all $i_p = q$, it runs the weigh-coordinates algorithm on the $(\vec{i}-\vec{e}_p)$th evaluation subarray, $\{f_{\vec{i}-\vec{e}_p+\vec{e}_k}(u) : k = 0..d\}$. The algorithm accumulates the contributions of $f'_{\vec{i}}(u)$ to each value,
>
> $$f_{\vec{i}-\vec{e}_p+\vec{e}_k}(u) \longleftarrow f_{\vec{i}-\vec{e}_p+\vec{e}_k}(u) + f'_{\vec{i}}(u)\lambda^{\vec{i}+\vec{e}_p-\vec{e}_k}_k$$
>
> yielding the desired values of $f_{\vec{i}}(u)$ in Equation 4.4.

Knot Replacement

Next, I present the relation between the basis B and the basis where the knot $t_{p,n-1}$ of B is replaced by a new point x. This relation is obtained by exactly the same arguments as for

knot swapping.

$$t'_{\vec{i}} = \begin{cases} t_{\vec{i}-\vec{e}_p} x, & \text{if } \vec{i} = (n-1)\vec{e}_p \\ t_{\vec{i}}, & \text{otherwise} \end{cases}$$

The only difference between this equation and Equation 4.3 is that there are no $\vec{i}$ where $i_p > n-1$.

> The *knot replacement* algorithms compute new coefficients and coordinates over the new basis. They operate in a completely analogous way to the knot swapping algorithms.

Swapping and Replacing Successively

The knot swap and knot replacement algorithms can be run successively to move a knot several positions, or to introduce a new knot into an arbitrary position (p, q).

If knot swapping is run first for $i_p = q$, then $i_p = q+1$, until $i_p = q+k-1$, it will move knot $t_{p,q}$ to $(p, q+k)$ and move all the knots $t_{p,q+1}, \ldots, t_{p,q+k}$ backward one position to $(p,q), \ldots, (p, q+k-1)$. Conversely, running knot swapping for $q-1$, $q-2$, until $q-k$, will move the knot $t_{p,q}$ to $(p, q-k)$ and move the the knots $t_{p,q-1}, \ldots, t_{p,q-k}$ forward one position.

Running knot replacement puts the new knot x at $(p, n-1)$. This operation can be thought of as swapping the knot "the nth knot" $t_{p,n} = x$ to $(p, n-1)$. If this operation is followed by several knot swaps, the new knot x can be put in any position (p, q). The resulting algorithm is a generalization of Boehm's knot insertion algorithm [3, 4] for B-splines.

Figures 4.4 and 4.5 show examples of replacing and swapping knots. In Figure 4.4, the bottom triangular arrays holds the original coefficients. The middle left array holds the coefficients of the knot net where $t_{1,2}$ is replaced with x. All the coefficients are the same as the bottom array except the one indicated (at the lower left corner); that coefficient is computed by a combine-coefficients operation. The top left array shows the knots x and $t_{1,1}$ being swapped. Again, all coefficient are the same, except two. On the right, the operations are performed in reverse order: first swap, then replace. Notice that on the left, the second operation uses only original coefficient, whereas on the right, the second operation uses coefficients that result from the first operation. Thus, on the right, there is potential for round-off errors to build up.

In Figure 4.5, the top arrays hold the original coordinates. Again, each new array keeps the same coordinates as the previous, except for the ones indicated; those coordinates are computed with a weigh-coordinates operation. Again, the left side shows a knot replace followed by a knot swap, while the right side shows a knot swap followed by a knot replace. In this figure, there is potential for round-off errors to build up on the left side.

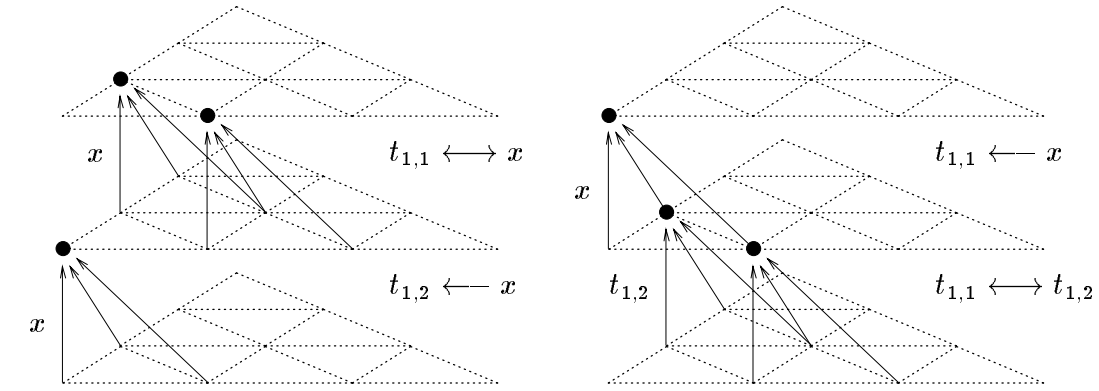


Figure 4.4: Knot swapping for coefficients

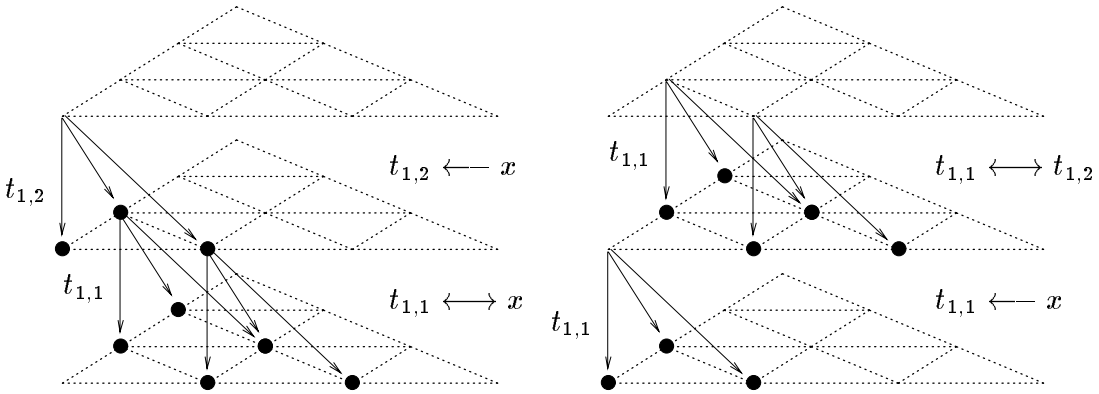


Figure 4.5: Knot swapping for coordinates

4.5 Run-time Analysis of Algorithms

Combining and Weighing

Let the domain be dimension d, the range be dimension d', and the polynomial be degree n.

The cost of combine-coefficients is the cost of getting coordinates plus combining plus indexing. Getting the coordinates of a domain point requires solving a system of linear equations, and requires normally $O(d^3)$ operations. Taking linear combination of $d+1$ ranges points, if done coordinate-wise, costs $O(dd')$ operations. Thus a single combine-coefficients costs $O(d^3 + dd')$ in total. Weigh-coordinates multiplies and adds to d' scalars, which costs $O(d')$. Thus a single weigh-coordinates costs $O(d^3 + d')$ in total.

Note that for Bézier bases, the $\vec{i}$th evaluation basis is the same for all $\vec{i}$. Thus, they only require getting the coordinates once, and those coordinates can be used for all calls to combine-coefficients or weigh-coordinates. For Bézier bases, the cost of combine-coefficients or weigh-coordinates are only $O(dd' + d^2)$ and $O(d' + d^2)$. (The d^2 comes from the cost of indexing with multi-indices: to convert a multi-index to a linear index requires $O(d)$

operations, and this conversion is done d times. In the general case, this term is dominated by the cost of getting evaluation basis coordinates.)

Because the operation count depends on the basis, the run-time analysis of the remaining algorithms determine how many combine-coefficients or weigh-coordinates the algorithm requires rather than how many arithmetic operations.

Evaluation and Getting Coordinates

One level of partial evaluation performs combine-coefficients for all $\vec{\imath} \in \mathbb{I}_d^{n-1}$, so it costs $D(d, n-1)$ combine-coefficients. For k levels, it costs

$$D(d, n-1) + \cdots + D(d, n-k) = D(d+1, n) - D(d+1, n-k)$$

combine-coefficients. Thus, a full evaluation costs $D(d+1, n-1)$ combine-coefficients. Getting coordinates requires the same number of weigh-coordinates.

Knot Swapping

To swap one knot, the algorithm performs combine-coefficients or weigh-coordinates for all multi-indices $\vec{\imath}$ such that $i_p = q$. There are $D(d-1, n-q)$ of these multi-indices. To move a knot from q to $q+k$, then, requires

$$D(d-1, n-q) + \cdots + D(d-1, n-q-k) = D(d, n-q) - D(d, n-q-k) + D(d, q+k)$$

combine-coefficients or weigh-coordinates.

4.6 Round-off Error Accumulation

In a sequence of computations, when the results of one blossom operation is used in the next operation, there is a potential for round-off errors to accumulate. A useful measure of this potential is how many combine-coefficients or weigh-coordinates operations are successively applied to a value to obtain the result. Thus, the *depth* of a computation is defined to be maximum number of combine-coefficients or weigh-coordinates operations separating a resulting value of the computation from the original values.

For one level of partial evaluation or getting partial coordinates, all computations use original values. Thus, the depth is one. The next level would use the new values. Thus for k levels, the depth is k. For a full evaluation or getting full coordinates, the depth is n. For knot swap or knot replacement, the depth is one.

When successively swapping a knot several positions, the depth depends on which direction the knot is moved:

- for coefficients, swapping a knot forward from q to $q-k$ has depth k, while swapping a knot backward from q to $q+k$ has depth 1.

- for coordinates, swapping a knot forward from q to $q-k$ has depth 1, while swapping a knot backward from q to $q+k$ has depth k.

This behaviour is shown in Figures 4.4 and 4.5. The left side of Figure 4.4 shows that when swapping the new knot x to forward position $(1,1)$, the algorithm always uses original values, so the depth is one. On the other hand, the right side shows that when swapping the knot $t_{1,1}$ backward, each step uses the results of the previous step, so the depth is two. Figure 4.5 shows the opposite happens when swapping a knot for coordinates. When swapping x forward to $(1,1)$, the later steps use results from the earlier steps, while when swapping $t_{1,1}$ backward, each step uses original values.

Chapter 5

Implementation

The biggest challenge in the implementation was to make the library work with many different types. To work with different types means the library accesses all objects through an abstract interface. The first issue is how to implement abstract interfaces in C++. The second issue is what functions should be included in the abstract interface for each datatype.

5.1 Templates vs. Inheritance

I decided to use the C++ template facility to implement the abstract interfaces of datatypes. The alternative of using inheritance was ruled out based on three criteria: the ease of integrating a type with the library, the ease of specializing operations and datatypes of the library, and efficiency of the compiled code.

With the template approach, arbitrary types can be used in a class or function. The code of the class or function accesses objects of the datatype through the functions listed in the abstract interface. Any attempt to use a type that does not provide the necessary functions results in a compiler error. With the inheritance approach, an abstract interface is defined as a set of virtual member functions of a superclass. The datatype is actually implemented by subclasses which override the virtual functions. A subclass may choose to "inherit" certain virtual functions form the superclass rather than overriding them.

Integrating a type into the library is easier when non-member functions are required than when member functions are required, because non-member functions can be created for built-in types or library classes whose source code is unavailable. Templates allow either member functions or non-member functions to be part of the abstract interface, while inheritance only supports abstract member functions.

Specializing a function under the inheritance approach is accomplished by by overriding virtual functions in subclasses. The overriding facility allows the programmer to specialize a function based on the type of *one* object. However, inheritance has no support for specializing based on the types of *two or more* arguments, so-called multi-methods. Templates can be

specialized for any number of arguments. Multi-methods are useful, for example, when evaluating Bézier curves. These blossoms have one-dimensional triangular arrays and Bézier knot nets.

Finally, since the library provides low-level computational objects, the objects need to be highly efficient for some applications. Templates introduce no overhead at run-time, while virtual functions do. More importantly, templates allow inlining, while virtual functions do not. With templates, the compiler decides at compile-time which implementation to use, while with virtual functions, the decision is made at run-time. Since the library uses abstract functions, it requires a lot of function calls. Thus inlining of functions is very important.

5.2 Abstract Interface

The second issue involves choosing what functions to require in the abstract interface of each datatype. The choice is driven by what is needed to implement the blossom operations. When there is a choice between two ways to implement the operations, the following guidelines are followed.

Since the library must be general, the abstract interface should include as few requirements as possible. The requirement should be based on fundamental mathematical properties of the objects—the lowest common denominator.

Certain steps in operations can be more efficiently implemented for some special cases than in general. For such steps, the operation leaves the actual implementation of that step for the datatype. That is, a function is required of the datatype that performs that step. As a result of this approach, the blossom operations are very simple and contain only a few lines of code. They make calls to the abstract functions to do most of the work.

Finally, the abstract interface should avoid requiring member functions in favor of non-member functions. Where applicable, the abstract interface should use STL iterators and containers.

5.3 Blossom Operations

The implementation involves three parts. The first involves coding the blossom operations. The operations determine what functions need to be defined in the abstract interface of the blossom datatypes. The second part is to code the blossom datatypes. The datatypes, in turn, determine what to require of the geometry datatypes. The last part is to code geometry datatypes. This section discusses the coding of the blossom operations, which drives the whole of the implementation.

All blossom operations are built on the two core operations, combine-coefficients and weigh-coordinates. The pseudo-code for combine-coefficients is obtained by expanding the algorithm given in Section 4:

Given a knot net, $\{t_{k,l} : k = 0..d, l = 0..n-1\}$, a triangular array of coefficients, $P_{\vec{i}}$, a multi-index, $\vec{i} \in \mathbb{I}_d^{n-1}$, and a domain point x,
1. extract the $\vec{i}$th evaluation basis of the knot net, $\{t_{0,i_0}, \ldots, t_{d,i_d}\}$.
2. compute the coordinates $\lambda_k^{\vec{i}}$ of x over the basis.
3. extract the $\vec{i}$th evaluation subarray of the triangular array, $\{P_{\vec{i}+\vec{e}_k}, k = 0..d\}$.
4. return the combination of the coordinates and the coefficients, $\sum_k \lambda_k^{\vec{i}} P_{\vec{i}+\vec{e}_k}$.

The pseudo-code for weigh-coordinates is as follows:

Given a knot net, $\{t_{k,l} : k = 0..d, l = 0..n-1\}$, a triangular array of coordinates, $\alpha_{\vec{i}}$, a multi-index, $\vec{i} \in \mathbb{I}_d^{n-1}$, a domain point x, and a coordinate α,
1. extract the $\vec{i}$th evaluation basis of the knot net, $\{t_{0,i_0}, \ldots, t_{d,i_d}\}$.
2. compute the coordinates $\lambda_k^{\vec{i}}$ of x over the basis.
3. extract the $\vec{i}$th evaluation subarray of the triangular array, $\{\alpha_{\vec{i}+\vec{e}_k}, k = 0..d\}$.
4. adds α times the coordinate to the weight of the new coordinates,

$$\alpha'_{\vec{i}+\vec{e}_k} \longleftarrow \alpha'_{\vec{i}+\vec{e}_k} + \alpha\lambda_k^{\vec{i}}.$$

The other blossom operations call these two operations with the appropriate multi-indices. The evaluation and getting coordinate operations call them for every multi-index $\vec{i} \in \mathbb{I}_d^n$, while the knot swapping operations call them for all $\vec{i} \in \mathbb{I}_d^n$ such that $i_p = q$.

The next sections discuss what functions and types are required to implement these operations. Only the interesting functions are discussed here. The full requirements are listed in Appendix A.

5.4 Blossoming Datatypes

5.4.1 Blossom

Although the operations work naturally with knot nets and coefficient arrays, for the convenience of the user the library provides a blossom datatype. A blossom type simply needs to provide functions to return the knot net and the coefficient array. In addition, since the partial evaluation operations return a blossom object, a blossom type needs to provide a function that can create blossoms of that type from arbitrary knot nets and coefficients.

5.4.2 Knot Net

The main requirement of a knot net is to supply an evaluation basis given a multi-index. Note that for Bézier knot nets, the evaluation bases—and the coordinates—are the same for all multi-indices $\vec{i}$. Thus, when the combine-coefficients or weigh-coordinates operations are called many times, it is more efficient to compute the coordinates once, save them, and use them in all combinations. It is the responsibility of the knot net class to decide how

it computes the coordinates. The class provides two types, `combiner` and `weigher`, and two functions, `combineCoeffs` and `weighCoords`. The `combiner` and `weigher` objects for a Bézier knot net compute and store the coordinates when they are created, whereas for a general knot net, they only allocate memory to store the coordinates. The `combineCoeffs` and `weighCoords` use the `combiner` and `weigher` objects to either get the coordinates or store the coordinates.

In addition, non-Bézier knot nets implement `swapKnot` and `setKnot` functions.

5.4.3 Triangular Arrays and Multi-indices

The main requirement of triangular arrays is to return the $\vec{i}$th element and the $\vec{i}$th evaluation subarray. I decided to keep the triangular array datatype simple and put the complexity into the multi-index datatype. The triangular array datatype simply acts like a linear array, that is, it can return an element given an integer. The multi-index datatype has to compute which integer corresponds to a multi-index $\vec{i}$, and which integers correspond to the $\vec{i}$th evaluation subarray. However, the triangular array still needs to provide its dimension and degree, since blossom operations need that information.

The operations of evaluation and getting coordinates need some way to iterate over all multi-indices $\vec{i} \in \mathbb{I}_d^n$. Thus the multi-index type must provide functions to return the first multi-index in the set, increment to the next multi-index, and test if the last multi-index has been reached. The ordering for the multi-indices is unspecified, although the most popular ordering is the reverse lexicographical. Similarly, knot swapping operations requires iterating over all multi-indices $\vec{i} \in \mathbb{I}_d^n$ such that $i_p = q$. This is called a *slice* of a triangular array. The same kind of functions must be provided to iterate over a slice.

Finally, a blossom operation must decide which multi-index type to use, based on its parameters of a knot net and an input triangular array and possibly an output array. For example, if the array class is of fixed dimension, it is more efficient to use a fixed-size multi-index class instead of a general multi-index class. It is inefficient to use three multi-index objects and increment them in-step; moreover, the orderings used by all three multi-index types must be the same.

I decided to have the input array specify the multi-index type (`triarray::multiindex`). This decision was made because the input array has to compute the evaluation subarrays, which is a procedure that can be optimized [20]. The output triangular array and the knot net access the multi-index type through functions that return the multi-index's components and position in the ordering.

5.5 Geometric Datatypes

5.5.1 Domain

A domain space is required to provide coordinates of a point over a basis. Thus, it requires a `basis` and a `point` type. The `basis` class should provide a way to create a basis (for creating evaluation bases) and to modify elements in a basis (for Bézier bases).

In addition, the blossom operation must store the coordinates somewhere. If the domain space has variable dimensions, a dynamically allocated array is required to hold the coordinates, but if it is fixed dimensional, a fixed size array can hold the coordinates, and dynamically allocating an array is inefficient. Thus, a `domain` class is required to specify a `domain::coordarray` type to hold the coordinates. The `coordarray` information is not specified by `point` or `basis` because those may be existing types and the user may be unable to modify them.

Blossoms and knot nets classes need to know how big an array to create, so the dimension of the domain space must be accessible. Further, knot nets require some initial knots, and a standard basis is a good choice. These items can be packaged in a `space` type.

5.5.2 Range

A range space is required to take combinations of points. Thus, only a `rangept` class is required. The range space is required to handle affine combinations, and may handle linear combinations; if the range type does not handle linear combinations, it is up to the user of the operations to call the operations with arguments that are only points.

The range's combination function must take arbitrary iterators returning scalars and range points. Thus, the operation must have an extra parameter to specify which combination function to call. The combination operation uses the STL approach of using the `value_type()` function of the iterator, which returns a `rangept*`. Thus, the declaration of the combination function must be

```
template<scalarIterator, ptIterator>
rangept combination(scalarIterator begin, scalarIterator end,
                    ptIterator ptbegin,
                    rangept*);
```

The function must use the same scalar type as the domain.

5.5.3 Scalar

The `weighCoords` function requires scalars to be multiplied together and added to another scalar. Also, getting coordinates initializes the coordinates to zero or one, so the scalar type should be constructible from 0 and 1: `scalar(0)`, `scalar(1)`.

5.6 Classes

The implementation of the actual classes is straight-forward. The emphasis is on generality rather than efficiency. Thus, the classes `PtDomain`, `TriArray`, `MultiIndex` and `KnotNet` all use dynamically allocated arrays. This allows them to handle arbitrary dimensions and degrees.

In contrast, the `DoubleDomain` class handles the special case of one-dimensional affine domain spaces. It uses the built-in type `double` as the point type, and it is very efficient. The class is intended as a test of how simple it is to integrate specialized datatypes into the library.

The actual classes provide more functionality than just the requirements in the datatype interface. These additional functions make the classes more useful to the user. For example, there are useful constructor functions, and arithmetic operator for `MultiIndex` and `Pt`.

The next chapter provides examples of the use of these classes.

Chapter 6

Evaluation of the System

This chapter aims to evaluate the Blossom Classes library's performance in practice. It demonstrates the use of the Blossom Classes in a variety of situations. The first example uses the Blossom Classes for commonly performed computations, showing that the library is easy to use. The second example creates specialized data structures, showing how well they integrate with the Blossom Classes. The last three examples implement algorithms for new operations, demonstrating that the library is useful for research. Each example examines the ease of translation from analysis to code, and evaluates the resulting code for efficiency and potential to accumulate round-off error.

6.1 Simple Demonstration

Shaping and Tesselating a B-patch

These first examples use Blossom Classes to do common operations. This example below uses the library to make a B-patches with different knots and coefficients. It demonstrates the three common operations of creating a blossom, setting knots and coefficients, and evaluating.

```
1    #include <iostream.h>
     #include <stl.h>
     #include "Blossom/Blossom.h"
     #include "Blossom/BlossomOps.h"
5    #include "Blossom/PtDomain.h"

     void tesselate(const Blossom<PtDomain, Pt> &f) {
       int n = f.getDegree();
       for (double x=0;x<=1.0;x+=.1) {
10       for (double y=0;y<=1.0-x;y+=.1) {
           cout << diagonalEval(f,Pt(x,y,1)) << endl;
         }
```

```
      }
      for (MultiIndex i=firstTriIndex(2,n);;) {
15      cout << f.getCoeff(i) << endl;
        if (isLastTriIndex(i,2,n)) break;
        nextTriIndex(i);
      }
    }
20
    void main()
    {
      Blossom<PtDomain, Pt> f(getStdSimplex(PtSpace(2)),2);
      Pt coeffs[] = { Pt(0,0,0,1), Pt(1,0,0.5,1), Pt(2,0,0,1),
25                    Pt(0,1,0.5,1), Pt(1,1,1,1), Pt(0,2,0,1)};
      copy(coeffs, coeffs+6, f.getCoeffs().begin());
      tesselate(f);
      f.setCoeff(E(0)+E(2),Pt(.5,1.5,.5,1));
      tesselate(f);
30    f.setKnot(0,1,Pt(1,.5,1));
      tesselate(f);
    }
```

The main routine first creates a blossom in line 23: the first argument to the constructor specifies the blossom should initially use a Bézier knot net over the standard simplex in 2-space, $\{(1,0,1),(0,1,1),(0,0,1)\}$, and the second argument specifies the degree of the blossom, quadratic. Lines 24–26 set up the initial coefficients (points in 3-space), and line 27 calls the tesselate function. Lines 28–31 demonstrate moving a coefficient and a knot. The expression `E(0)+E(2)` creates a `MultiIndex` object corresponding to the multi-index $\vec{e}_0+\vec{e}_2 = (1,0,1)$.

The `tesselate` function (lines 7–19) iterates over a tesselation of the standard simplex (in increments of .1). It prints out the computed value of the function at each point (line 11). Finally, it uses the `MultiIndex` class to iterate over all elements of the coefficient array, and prints them (lines 14–18).

Compute Position and Normal

The example below extends the `tesselate` function of the previous example to compute both the positions and normals of the surface. Evaluating a normal involves taking the cross product of two directional derivatives. From Equation 2.1, the derivative is proportional to $f(x^{n-1}\vec{v})$. Evaluating at the point x is $f(x^n)$. Thus, the partial evaluation $f|_{x^{n-1}}$ can be reused in the computation of the point and both directional derivatives.

```
1   void tesselate(const Blossom<PtDomain, Pt> &f) {
      int n = f.getDegree();
      Pt u(1,0,0), v(0,1,0);
```

```
      for (double x=0;x<=1;x+=.1) {
5       for (double y=0;y<=1-x;y+=.1) {
          vector<Pt> arg(n-1,Pt(x,y,1));
          Blossom<PtDomain,Pt> f2 = partialEval(f, arg.begin(), arg.end(),
                                                (Blossom<PtDomain,Pt>*)0 );
          Pt pos = diagonalEval(f2,arg[0]);
10        Pt norm = cross(diagonalEval(f2,u), diagonalEval(f2,v));
          cout << pos << pos+norm << endl;
        }
      }
    }
```

The following lines were changed from the previous function: line 3 defines the two vectors to evaluate directional derivatives; line 6 puts `n-1` copies of the evaluation point in an array; line 7 creates a blossom, `f2`, that is the partial evaluation of `f`; lines 9–11 uses the partial evaluation to evaluate the normal and the position.

The example programs output a list of points (and normals) on the surface. Figure 6.1 displays these points. In the three diagrams on the left, I joined adjacent points of the tesselation to make the shape of the patch easier to see.

Monomials

This example shows how to use the library for manipulating polynomials in the familiar monomial basis. It prints out values of the polynomial $x^3 - 2x^2 + 4x + 3$.

```
1   void main()
    {
      Blossom<PtDomain, Pt> f(getStdFrame(PtSpace(1)),3);
      Pt coeffs[] = { Pt(1,0), Pt(-2.0/3.0,0), Pt(4.0/3.0,0), Pt(3,1)};
5     copy(coeffs, coeffs+4, f.getCoeffs().begin());
      for (double x = 0;x<=1;x+=.1) {
        cout << diagonalEval(f,Pt(x,1)) << endl;
      }
    }
```

Line 3 creates a blossom whose knot net is the Bézier basis over the standard frame in one dimension, $\{\vec{\delta}, \mathbf{0}\}$. Lines 4 and 5 set the coefficients over this Bézier basis. By the relation given in Section 2.5.1, the coefficients are $\{\vec{\delta}, -\frac{2}{3}\vec{\delta}, \frac{4}{3}\vec{\delta}, \mathbf{3}\}$. As discussed in that section, every coefficient except the last is a vector.

Evaluation

As these three examples demonstrate, common operations can be performed easily, in the obvious way, and using an imperative coding style. The operations have the same run-time

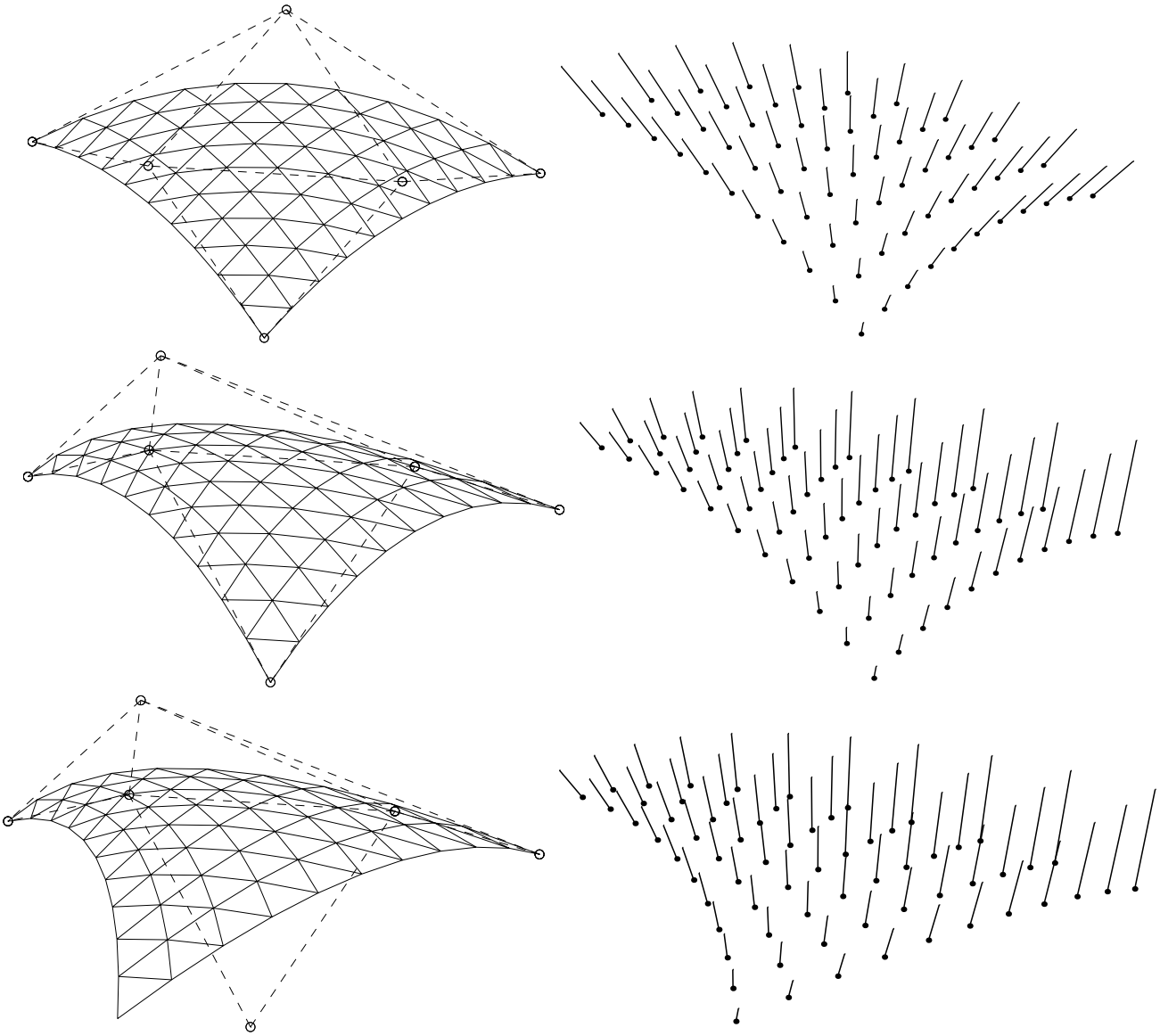


Figure 6.1: Results of simple example code.

and stability characteristics as the standard de Casteljau algorithm. Since the algorithms are for B-patches, they are a little less efficient than the de Casteljau algorithm for Bézier patches.

6.2 B-spline Datatype

Many modeling applications use piecewise polynomials, where each piece shares many coefficients and knots with its neighbor. Thus, it is inefficient to store a separate coefficient array and knot net for each piece. This examples shows how to extend the the Blossom Classes to handle piecewise polynomials. It demonstrates the creation of a B-spline curve datatype. It also shows how to specialize datatypes and algorithms for greater efficiency. Note that the same ideas can be used to implement a tensor-product patch datatype.

New Classes

The idea of the B-spline datatype is to create a new blossom datatype that corresponds to a segment of the B-spline. The segment datatype allows all the library's blossom operations to be used for B-splines. However, each segment does not store any of the coefficients or knots on it own. Rather the segment stores a pointer to the B-spline, and gets the knots and coefficients from the B-spline's knot vector and control vertices.

To keep the examples short, many of the functions for the classes are left out. The classes must provide all the functions required for the appropriate datatype listed in Appendix A.

The first class shown is the `Bspline` class, which stores a vector of knots and a vector of control vertices. The important function in the class is `getSegment`, which returns the ith segment of the B-spline.

```
1     class Bspline {
      private:
        vector<double> knots;
        vector<Pt> cvs;
5       int degree;
      public:
      ...
        inline Segment getSegment(int i) { return Segment(this,i); }
      ...
10    };
```

Note that, since its domain is one-dimensional, `Bspline` uses the built-in type `double` as the domain type.

The class `Segment` implements a new blossom datatype. The important functions for a blossom datatype are `getCoeffs` and `getKnotNet`. The special classes `OffsetArray` and `FoldKnots` get the segment's coefficients and knots from the B-spline.

```
1    class Segment {
     private:
       Bspline *f;
       int segment;
5    public;
       inline Segment(Bspline *f, int segment) : f(f),segment(segment) {}
     ...
       inline OffsetArray getCoeffs() { return OffsetArray(f,segment); }
       inline FoldKnots getKnotNet() { return FoldKnots(f,segment); }
10   ...
     };
```

From Section 2.6, the knot net of segment i is

$$\begin{pmatrix} t_{0,0} = t_{i+n-1} & \cdots & t_{0,n-1} = t_i \\ t_{1,0} = t_{i+n} & \cdots & t_{1,n-1} = t_{i+2n-1} \end{pmatrix}.$$

The functions in class `FoldKnots` return the knots from the B-spline knot sequence. For example, the `getEvalBasis` function below returns the $\vec{i}$th evaluation basis

$$\{t_{0,i_0}, t_{1,i_1}\} = \{t_{i+n-1-i_0}, t_{i+n+i_1}\}$$

```
1    class FoldKnots {
     private:
       Bspline *f;
       int segment;
5    public;
       inline FoldKnots(Bspline *f, int segment) : f(f),segment(segment) {}
     ...
       inline friend DoubleBasis getEvalBasis(SingleIndex i) {
         return DoubleBasis(f.knots[segment+f.degree()-1-i[0]],
10                          f.knots[segment+f.degree()+i[1]]);
       }
     ...
     };
```

The coefficients of segment i are $P_i, \ldots, P_{i+n}$. The `operator[]` function of the `OffsetArray` class returns these coefficients.

```
1    class OffsetArray {
     private:
       Bspline *f;
       int segment;
5    public;
       typedef SingleIndex multiindex;
```

```
    typedef vector<Pt>::iterator iterator;
    inline OffsetArray(Bspline *f, int segment) : f(f),segment(segment) {}
  ...
    inline Pt &operator[](int k) { return f->cvs[k+segment]; }
  ...
  };
```

Finally, since all arrays in the application are one dimensional, it is more efficient to use a fixed-size multi-index class. The multi-index can be implemented as a pair of `int`s.

```
  class SingleIndex {
    int one,two;
  public;
    inline SingleIndex(int i0,int i1) :one(i0),two(i1) {}
    inline int operator[](int k) {
      if (k==0) return one;
      else if (k==1) return two;
      else return 0;
    }
  };

  inline SingleIndex firstTriIndex(int d, int n) {
    return SingleIndex(n,0);
  }
  ...
```

B-spline Operations

With the segment datatype, implementation of B-spline operations becomes simple: first, find which segment to operate on; then, extract the `Segment` object; finally, apply blossom operations on it.

The following code for evaluating the B-spline performs the de Boor algorithm.

```
  Pt evaluate(const Bspline &f, double x) {
    Segment g = f.getSegment(f.whichSegment(x));
    return diagonalEval(g,x);
  }
```

The following code for inserting knots into the B-spline performs the Boehm knot insertion algorithm.

```
  void knotInsert(Bspline &f, double x) {
    int i = f.whichSegment(x);
    int n = f.getDegree();
    int mult = f.multiplicity(x);
```

```
5       insert(f.knots.begin()+i+2*n-1,1,f.knots[i+2*n-1]);
        insert(f.cvs.begin()+i+n,1,f.cvs[i+n]);
        Segment g = getSegment(i);
        knotReplaceCoeffs(g,1,x);
        knotSwapCoeffs(g,1,n-1,mult-1);
10    }
```

Lines 5 and 6 duplicate the last knot and control vertex of the `i`th segment. Then, lines 7–9 extract the `i`th segment and run knot swapping on it. These two operations have the effect of inserting knot `x` into position `i+n-mult` of the knot vector.

Evaluation

New datatypes integrate into the library fairly easily. The number of functions each datatype has to provide is only about 10 (see requirement in Appendix A), and each function tends to be very short.

Because I tried to make the requirements general, the built-in `int` type could not be used as the multi-index type for segments, even though it would be more efficient and convenient. The multi-index type must provide functions to return the components of the multi-index since it must work with other kinds of knot nets besides `FoldKnots`; `int`s cannot provide that.

The new classes are very efficient; they have almost no overhead in terms of extra storage required (only an extra pointer and integer), or processing (all the objects are fixed size and all the functions are inlined). Yet, they allow B-spline operations to be easily implemented using blossom operations on the segments. The resulting operations use the same algorithms as a hand-coded B-spline class would use, such as the de Boor algorithm or Boehm knot insertion.

6.3 Basis Conversion

This section demonstrates the use of the Blossom Classes to implement algorithms for new operations. This example implements an important operation: basis conversion. Basis conversion can be used to convert from a B-spline segment to a Bézier curve and vice versa. Other algorithms also use basis conversion, such as polynomial composition.

The basic problem is to compute the coefficients of f over the knot net $\{t'_{k,l}\}$ given its coefficients over the knot net $\{t_{k,l}\}$.

There are many ways to solve this problem which are mathematically equivalent, but result in algorithms that have different characteristics with respect to simplicity of code, run time and potential to accumulate round-off error. In the following sections, I compare two different solutions.

Basis-Conversion One: Change-of-Basis Matrix

One solution makes use of the get coordinates operation to compute the change of basis matrix between the two knot nets. For each $t'_{\vec{i}}$, compute its coordinates in the old basis: $t'_{\vec{i}} = \sum_{\vec{j}} \alpha_{\vec{i},\vec{j}} t_{\vec{j}}$. Then the new coefficients are $f(t'_{\vec{i}}) = \sum_{\vec{j}} \alpha_{\vec{i},\vec{j}} f(t_{\vec{j}})$.

```
1    void basisConvert1(Blossom<PtDomain,Pt> &f,
                        const KnotNet<PtDomain> &knots) {
       int d = getDimension(f.getSpace());
       int n = f.getDegree();
5      TriArray< TriArray<double> > coords(d,n,TriArray<double>(d,n,Pt(d)) );
       vector<Pt> basiselem(n,Pt(d));
       for (MultiIndex i=firstTriIndex(d,n);;) {
         getBasisElement(knots,i,basiselem.begin());
         getCoords(f.getKnotNet(),coords[i],basiselem);
10       if (isLastTriIndex(i,d,n)) break;
         nextTriIndex(i);
       }
       TriArray<Pt> newcoeffs(d,n,Pt(d));
       for (i=firstTriIndex(d,n);;) {
15       for (j=firstTriIndex(d,n);;) {
           newcoeffs[i] += coords[i][j] * f.getCoeff(j);
           if (isLastTriIndex(j,d,n)) break;
           nextTriIndex(j);
           }
20       if (isLastTriIndex(i,d,n)) break;
         nextTriIndex(i);
       }
       f = Blossom<PtDomain,Pt>(knots,newcoeffs);
     }
```

The code consists of two loops: the first loop (lines 7–12) computes the change-of-basis matrix, and the second loop (lines 13–22) uses the matrix to compute the new coefficients. Line 5 sets up a triangular array of $D(d,n)$ triangular arrays to store the change-of-basis coordinates. Line 6 sets up a vector to store the basis tensors. Line 8 puts the `ith` basis tensor into the vector. Line 9 stores the coordinates of the basis tensor in the `ith` coordinate array. Line 13 sets up an array of new coefficients. Line 16 multiplies the coordinates and the coefficients of `f` and adds them to the new coefficients. Line 23 sets `f`'s knots and coefficients.

As this code requires a full get-coordinate operation for each basis element, its run-time is $O(D(d+1,n-1)D(d,n))$. In terms of round-off error accumulation, the code behaves fairly well: the coordinates require n weigh-coordinates and each new coefficient is computed from one linear combination of those coordinates. Thus, the total depth of the new coefficients is $n+1$.

Basis-Conversion Two: Knot Swapping

Another solution uses knot swapping. The idea is simply to replace $t_{k,l}$ with $t'_{k,l}$ for all k and l. This following code is a generalization of Goldman's basis conversion algorithm for local B-spline bases [12]. If the knot nets are Bézier, the code is simply performing repeated subdivision.

```
1    void basisConvert2(Blossom<PtDomain,Pt> &f,
                        const KnotNet<PtDomain> &knots) {
       int d = getDimension(f.getSpace());
       int n = f.getDegree();
5      for (k=0;k<d;k++)
         for (l=0;l<n;l++) {
           knotReplaceCoeffs(f,k,knots.getKnot(k,l));
           knotSwapCoeffs(f,k,n-1,l);
         }
10   }
```

This code is much simpler than the previous one. The code in lines 7 and 8 puts the knot `(k,l)` of the new knot net into the `(k,l)` position of `f`'s knot net, and remove the old knot from `(k,n-1)`. This is done for all `(k,l)`.

This code is also much faster than the previous. The cost of `knotReplaceCoeffs` is one combination, while the cost of swapping the knot to position `(k,l)` is $D(d,l) - D(d,0)$. Adding up this cost for all `l`, it becomes

$$n + \sum_{l=0}^{n-1} D(d,l) - nD(d,0) = D(d+1, n-1),$$

or equivalent to the cost of a full evaluation. Thus, in total, the cost is $O((d+1)D(d+1, n-1))$, compared to $O(D(d+1, n-1)D(d,n))$.

However, the code is more likely to suffer from accumulation of round-off errors. Since results of earlier knot swaps are used for later ones, the depth of resulting coefficients gets as high as $(d+1)(n-1)$. Moreover, the code as presented will not always work. It may happen, as the knots get replaced, that swapping or replacing some knot `(d,k)` causes an evaluation basis to become degenerate. Then the knot net is no longer valid, and any attempt to use it will likely cause division-by-zero errors. A smarter implementation would need to check for this condition. However, if it is known that the knots $\{t_{k,l} : l = 0..n-1\}$ and the knots $\{t'_{k',l} : l = 0..n-1\}$ are enclosed in disjoint circles for all k and k', then this problem will not occur.

Finally, note that the code could have been written in several different ways. For example, lines 6–9 can be rewritten

```
       for (l=n-1;l>=0;l--) {
         knotReplaceCoeffs(f,k,knots.getKnot(k,l));
```

```
    knotSwapCoeffs(f,k,n-1,0);
}
```

This code puts the first new knot into position `(k,0)` of `f`'s knot net. Then, it puts the next new knot into position `(k,0)`, which moves the previously inserted knot into `(k,1)`, and so on. This code is less efficient because the knot manipulation operations require more computation for knots at low positions than for knots at higher positions. Also, this code is more likely to accumulate round-off errors because new values computed for the coefficients for the first substitution `(k,n-1)` are used to compute the the second substitution `(k,n-2)`, and so on. This increases the depth of resulting coefficients.

Evaluation

This example shows that it is easy to translate from analysis into code. It also shows that while there are many mathematically equivalent ways to achieve the same result, each way leads to a different implementation. Knowledge of how the operations work helps to create efficient algorithms that keep round-off errors under control.

6.4 Polynomial Composition

This section provides another example of implementing a new algorithm, the composition of Bézier simplices. DeRose et al [8] derived a formula for computing the Bézier control points of the composition polynomial $H(x) = F(G(x))$.

Let the degree of F be m and degree of G be ℓ. Let I be a vector of m multi-indices, $I = (\vec{i}_1, \ldots, \vec{i}_m)$, where each $\vec{i}_k$ has degree ℓ and dimension d. I is called a *hyperindex*. Let $|I| = \vec{i}_1 + \cdots + \vec{i}_m$. Then the control points of H are related to the control points of G by the following equation:

$$H_{\vec{i}} = \sum_{|I|=\vec{i}} C(I) f(G_{\vec{i}_1} \cdots G_{\vec{i}_m}) \text{ where } C(I) = \frac{\binom{|\vec{i}_1|}{\vec{i}_1} \cdots \binom{|\vec{i}_m|}{\vec{i}_m}}{\binom{||I||}{|I|}}. \tag{6.1}$$

In the following sections, I give several algorithms for computing the $H_{\vec{i}}$, starting with a direct translation, and then successively refining the algorithm to improve its efficiency. These algorithms are analyzed in Mann and Liu's paper [15]. I show how the library supports research into algorithms by providing a direct translation from efficient algorithms into efficient C++ code.

Composition One: Direct Translation

Equation 6.1 can be directly translated to code: iterate over all $\vec{i}$ and all I, evaluating $C(I)f(G_{\vec{i}_0}, \ldots, G_{\vec{i}_m})$:

```
1    #include "HyperIndex.h"
     long C(const HyperIndex &I);

     Blossom<PtDomain,Pt>
5    composition1(const Blossom<PtDomain,Pt>&  f,
                  const Blossom<PtDomain,Pt>&  g)
     {
       int d = getDimension(g.getSpace());
       Blossom<PtDomain,Pt> h(g.getDomainBasis(),
10                            f.getDegree()*g.getDegree());
       vector<Pt> args(d,Pt(d));
       for (MultiIndex i = firstTriIndex(d,h.getDegree());;) {
         for (HyperIndex I = firstHyperIndex(i);;) {
           for (int k=0;k<d;k++)
15           args[k] = g.getCoeff(I[k]);
           h.getCoeff(i) += C(I) * eval(f,args.begin());
           if (isLastHyperIndex(I,i)) break;
           nextHyperIndex(I);
         }
20       if (isLastTriIndex(i,d,h.getDegree())) break;
         nextTriIndex(i);
       }
     }
```

The code for the `HyperIndex` class and the function `C(I)` are assumed to exist.

Composition Two: Using Symmetry

The above code is inefficient because f is symmetric. Thus, the order of the $\vec{i}_k$ in I does not matter; it only how many times a multi-index $\vec{i}$ appears in I matters. The new algorithm evaluates f only once for all hyperindices that contain the same multi-indices appearing the same number of times. The computed value must be multiplied by a factor, $P(I)$, that indicates how many of these hyperindices there are.

Choose some ordering on $\mathbb{I}_d^n$. Let $\vec{0}, \vec{1}, \vec{k}$ be the first, second, kth multi-indices in this ordering. Then, the algorithm evaluates the values

$$P(\vec{i})C(\vec{i})f(G_{\vec{0}}^{i_0}, G_{\vec{1}}^{i_1}, \ldots, G_{\overrightarrow{D(d,\ell)-1}}^{i_{D(d,\ell)-1}})$$

for $\vec{i} \in \mathbb{I}_{D(d,\ell)}^m$. That is, the algorithm computes the value of f once for all hyperindices in which $\vec{0}$ appears i_0 times, $\vec{1}$ appears i_1 times and so on. The following code iterates over multi-indices in $\mathbb{I}_{D(d,\ell)-1}^m$ and computes the values of f.

```
1    long C(const MultiIndex &i);
     long P(const MultiIndex &i);
```

```
    Blossom<PtDomain,Pt>
5   composition2(const Blossom<PtDomain,Pt>&  f,
                 const Blossom<PtDomain,Pt>&  g)
    {
      int d = getDimension(g.getSpace());
      Blossom<PtDomain,Pt> h(g.getDomainBasis(),
10                           f.getDegree()*g.getDegree());
      int D = triSize(d,g.getDegree());
      vector<Pt> args(d,Pt(d));
      for (MultiIndex i = firstTriIndex(D-1,f.getDegree());;) {
        vector<Pt>::iterator p = args.begin();
15      MultiIndex sum;
        for (int k=0;k<D;k++) {
          for (int l=0;l<i[k];l++) {
            MultiIndex ik = OrdToIndex(i[k]);
            sum += ik;
20          *p++ = g.getCoeff(ik);
          }
        }
        h.getCoeff(sum) += P(i) * C(i) * eval(f, args.begin());
        if (isLastTriIndex(i,D-1,h.getDegree())) break;
25      nextTriIndex(i);
      }
    }
```

The loop in lines 15–22 converts the multi-index `i` into the corresponding argument for the blossom:

$$G_{\vec{0}}^{i_0}, G_{\vec{1}}^{i_1}, \ldots, G_{\overrightarrow{D(d,\ell)-1}}^{i_{D(d,\ell)-1}}.$$

The multi-index `sum` stores the position of corresponding coefficient of `H`.

Composition Three: Reusing Intermediate Calculations

The previous code is still inefficient because to compute $f(G_{\vec{i}_1}, \ldots, G_{\vec{i}_m})$, the partial evaluations for $f|_{G_{\vec{i}_1}}$ to $f|_{G_{\vec{i}_1} \cdots G_{\vec{i}_{m-1}}}$ must be computed. These partial evaluations can be reused for different I.

DeRose et al presented an algorithm that reuses the partial evaluations. The algorithm proceeds by partially evaluating at some $G_{\vec{\imath}}$. Then for all $\vec{\jmath} > \vec{\imath}$, recursively evaluate at $G_{\vec{\jmath}}$. Continue this process until f is fully evaluated. Then go back and select the next multi-index following $\vec{\imath}$. The library allows their pseudocode to be directly translated into C++:

```
1   void
```

```
PostProcessH(Blossom<PtDomain,Pt> &h)
{
  int d = getDimension(h.getSpace());
  int n = h.getDegree();
  for (MultiIndex j=firstTriIndex(d,n);;) {
    h.getCoeff(j) /= multinomial(j);
    if (isLastTriIndex(j,d,n) break;
    nextTriIndex(j);
  }
}

void
RecursiveCompose3(const Blossom<PtDomain,Pt>&   f,
                  const Blossom<PtDomain,Pt>&   g,
                  Blossom<PtDomain,Pt>&         h,
                  const MultiIndex&             min,
                  const MultiIndex&             sum,
                  long                          c,
                  long                          mu)
{
  if (f.getDegree() == 0) {
    h.getCoeff(sum) += c * *f.getCoeffs().begin();
  } else {
    MultiIndex i = min;
    Blossom<PtDomain,Pt> tmp = partialEval(f,g.getCoeff(i));
    RecursiveCompose3(tmp,g,h,i,sum+i,c*multinomial(i)/(mu+1),mu+1);
    while (!isLastTriIndex(i,getDimension(g.getSpace()),g.getDegree())) {
      nextTriIndex(i);
      tmp = partialEval(f,g.getCoeff(i));
      RecursiveCompose3(tmp,g,h,i,sum+i,c*multinomial(i),(long)1);
    }
  }
}

Blossom<PtDomain,Pt>
composition3(const Blossom<PtDomain,Pt>&  f,
             const Blossom<PtDomain,Pt>&  g)
{
  int d = getDimension(g.getSpace());
  Blossom<PtDomain,Pt> h(g.getDomainBasis(),
                         f.getDegree()*g.getDegree());
  MultiIndex zero,first = g.getDegree()*E(0);
  RecursiveCompose3(f,g,h,first,zero,factorial(f.getDegree()),(long)0);
```

```
45      PostProcessH(h);
        return h;
      }
```

Note that the algorithm was also optimized to reuse intermediate results to compute $C(I)$ (through the parameters `c` and `mu` of function `RecursiveCompose3`).

Composition Four: Changing Bases

The previous algorithm can be further refined to remove the remaining inefficiencies. The previous algorithm reuses the intermediate partial evaluations of f, but finally, those also get discarded. Mann and Liu [15] presented an algorithm that avoids discarding these evaluations. The algorithm is asymptotically optimal.

The new algorithm first converts the basis of F to a domain simplex that is a subset of G's coefficients, say $G_{\vec{i}_0}, \ldots, G_{\vec{i}_d}$. Then, the new coefficients of F are of the form $\{f(G_{\vec{i}_0}^{j_0} \cdots G_{\vec{i}_d}^{j_d})\}$, which are values of f needed to compute H's coefficients. Starting with these coefficient of F, the rest of the algorithm runs in a similar manner to the algorithm of DeRose et al, with the difference that all the intermediate points calculated contribute to the final result, and no calculations are thrown away (except the initial basis conversion). The following code is a direct translation of their pseudo-code:

```
1     void
      RecursiveCompose4(const Blossom<PtDomain,Pt>&   f,
                        const Blossom<PtDomain,Pt>&   g,
                        Blossom<PtDomain,Pt>&         h,
5                       const MultiIndex&             min,
                        const MultiIndex&             sum,
                        long                          c,
                        long                          mu)
      {
10      ExtractCPs(f,h,min,sum,c);
        if (f.getDegree() == 0) {
          return;
        } else {
          MultiIndex i = min;
15        BlossomArg<PtDomain> u(g.getCoeff(i));
          Blossom<PtDomain,range> tmp = partialEval(f,u);
          RecursiveCompose2(tmp,g,h,i,sum+i,c*multinomial(i)/(mu+1),mu+1);
          while (i[getDimension(g.getSpace())] != g.getDegree()) {
            nextTriIndex(i);
20          u.setArgument(0,g.getCoeff(i));
            tmp = partialEval(f,u);
            RecursiveCompose2(tmp,g,h,i,sum+i,c*multinomial(i),(long)1);
          }
```

```
    }
}

Blossom<PtDomain,Pt>
composition4(const Blossom<PtDomain,Pt>&  f,
             const Blossom<PtDomain,Pt>&  g)
{
  int d = getDimension(g.getSpace());
  Blossom<PtDomain,Pt> h(g.getDomainBasis(),
                         f.getDegree()*g.getDegree());
  Blossom<PtDomain,range> f2 = f;
  KnotNet<PtDomain> knots(makeBasis(g.getCoeffs().begin(),g.getSpace()));
  basisConvert(f2,knots);
  MultiIndex zero, first = OrdToIndex(d);
  RecursiveCompose4(f2,g,h,first,zero,factorial(f2.getDegree()),(long)0);
  PostProcessH(h);
  return h;
}
```

The `ExtractCP` code is omitted to save space.

This code does not always work since it uses the first `d` coefficients of `g` as the new basis of `f2` and these coefficients may not form a basis of the domain. A more sophisticated implementation would need to handle this case.

The run time behavior of this algorithm depends on the algorithm used for `basisConvert` routine.

6.5 Degree-raising B-splines

Finally, I present a new algorithm to compute the degree-raised form of a B-spline efficiently. An efficient degree-raising algorithm has been known for Bézier curves, but not for B-splines.

The problem is to find the control points and knot sequence of the degree n version, G, of the degree $n-1$ B-spline F. These are obtained by noting that G and F have the same number of segments and the segments of G are degree-raised versions of the corresponding segments of F.

First, this relation gives G's knot sequence. It implies G must have the same breakpoints as F. (*Breakpoints* are knots that are the endpoints of an interval for some segment: t_{n-1} to t_{n+L}). Since G is one degree higher than F, but still has the same continuity, the multiplicity of each breakpoint in G's knot sequence must be one greater than the corresponding breakpoint in F's knots.

Next, I use the relation to derive G's control points. Let $\{t_0, \ldots, t_{2n-1+L}\}$ be G's knot sequence and $P_0, \ldots, P_{n+L}$ its control points. Let $g_{[t_i,t_{i+1})}$ and $f_{[t_i,t_{i+1})}$ be the blossoms of G

and F respectively over the interval $[t_{i+n-1}, t_{i+n})$, where $t_{i+n-1} < t_{i+n}$. Let $t_{(i)} \equiv t_i \cdots t_{i+n-1}$, and let $t_{(i)}/t_j \equiv t_i \cdots t_{j-1} t_{j+1} \cdots t_{i+n-1}$. The degree-raising formula for blossoms states

$$g_{[t_i,t_{i+1})}(t_{(i)}) = \frac{1}{n} \sum_{j=i}^{i+n-1} f_{[t_i,t_{i+1})}(t_{(i)}/t_j).$$

Equation 2.2 can be rewritten to yield a formula for P_i:

$$P_i \quad = \quad g_{[t_{i-1},t_i)}(t_{(i)}) = \cdots = g_{[t_{i+n-1},t_{i+n})}(t_{(i)})$$

This formula states that the blossoms of $n+1$ consecutive segments all have the same value at $t_{(i)}$. For knots that have multiplicity greater than 1, the corresponding segments do not exist. However, since the multiplicity of knots are less than $n+1$, there is always some segment $g_{[t_{k-1},t_k)}$ whose value at $t_{(i)}$ is P_i, where k is any number between i and $i+n$ such that $t_{k-1} < t_k$ and $k < n + L$.

Combining this formula with the previous one yields a formula for the G's control points in terms of blossom values of F's segments:

$$P_i = \frac{1}{n} \sum_{j=i}^{i+n-1} f_{[t_{k_j-1},t_{k_j})}(t_{(i)}/t_j), \tag{6.2}$$

where k_j are numbers between i and $i+n-1$ such that $t_{k_j-1} < t_{k_j}$ and $k_j < n + L$.

Algorithm

From Equation 6.2, one can compute the degree-raised control points simply by computing the required values of the blossom $f_{[t_{k_j-1},t_{k_j})}$ for each P_i. However, since successive control points will require many of the same blossom values, a more efficient algorithm would be to store each value computed and use it again for later control points. Even this approach is not as efficient as possible, since parts of the computation for one blossom value can be reused for subsequent blossom values.

The following algorithm is arranged in such a way as to maximize reuse of computations. Moreover, the algorithm is easy to understand because it is made up of knot insertions and knot deletions. I first deal with the case when F has no knots of full multiplicity, $n-1$. Later I will consider the case of knots of full multiplicity.

The algorithm proceeds as follows:

1. Insert knots into F so that the multiplicity of each break point is increased by one. Let $\{t_0, \ldots, t_{2n-1+L}\}$ be the resulting knot sequence. Let Q_i be the new control points of F. By Equation 6.2,

$$Q_i = f_{[t_{i-1},t_i)}(t_{(i)}/t_{i+n-1}) = \cdots = f_{[t_{i+n-2},t_{i+n-1})}(t_i/t_{i+n-1}).$$

2. Set the knots of G to be same as the new knot sequence of F. Set G's control points initially to zero: $P_0 = \cdots = P_{n+L} = 0$.

3. For each $j = 0..2n-1+L$:

   (a) create new B-spline F^j by deleting knot t_j from F. Thus, the knots of F^j are

   $$\{t_0, \ldots, t_{j-1}, t_{j+1}, \ldots, t_{2n-1+L}\}.$$

   Define

   $$\begin{aligned} Q_i^j &\equiv f_{[t_{i-1},t_i)}(t_{(i)}/t_j) = \cdots = f_{[t_{i+n-2},t_{i+n-1})}(t_i/t_j). \qquad i = j-n+1..j \\ Q_i^j &\equiv f_{[t_{i-1},t_i)}(t_{(i)}/t_{i+n-1}) = \cdots = f_{[t_{i+n-2},t_{i+n-1})}(t_i/t_{i+n-1}) = Q_i, \qquad \text{o/w} \end{aligned}$$

   Note that $Q_j^j = Q_{j+1}^j$. By construction, Q_i^j are just the control points of F^j, with one point duplicated.

   (b) For each $i = j-n+1..j$, add Q_i^j/n to P_i:

   $$P_i \longleftarrow P_i + \frac{Q_i^j}{n}.$$

   This step will compute

   $$\begin{aligned} P_i &= \frac{1}{n} \sum_{j=i}^{i+n-1} Q_i^j \\ &= \frac{1}{n} \sum_{j=i}^{i+n-1} f_{[t_{k_j-1},t_{k_j})}(t_{(i)}/t_j) \end{aligned}$$

   where k_j is the number of some segment, $i-1 \le k_j \le i+n-1$.

Note that in Step 3(a), if $t_j = t_{j-1}$, the knot t_j does not have to be deleted again: the control points of F^{j-1} will be the same as those of F^j and can be reused.

Now consider the case where some breakpoint t_i of F has multiplicity $n-1$. Theoretically, the algorithm cannot insert that knot again, as B-splines cannot have knots of multiplicity greater than the degree; that would mean a discontinuity in the B-spline. However, this is not a problem in practice as long as the control points Q_i and Q_{i+1} are never different, which is the case for this algorithm. The knot insertion operation would simply duplicate the control point Q_i, and the knot deletion algorithm would remove a duplicate. Another approach is to flag the knot but do not insert it. Later, in Step 3(a), any knots that are flagged are not deleted.

Implementation

This code uses the class `Bspline`. The class is assumed to have the operation `deleteKnot`.

```
1    Bspline degreeRaise(Bspline F) {
       int n=F.getDegree();
       int L=F.knots.size()-2*n;
       int last = L+n-2;
5      for (int k=n; k<=last; k++) {
         if (F.knots[k] != F.knots[k-1]) < n-1) {
           insertKnot(F,F.knots[k]);
           last++;
         }
10     }
       int d=getDimension(getSpace(F.cvs[0]));
       Bspline G(n+1, F.knots);
       n = n+1;
       L=G.knots.size()-2*n;
15     Bspline Fj = F;
       for (j=0; j<=2*n-1+L; j++) {
         if (j=0 || F.knot[j] < F.knot[j-1]) {
           Fj = F;
           deleteKnot(Fj,j);
20       }
         for (int i=max(j-n+1,0); i<=min(j,2*n+L-2); i++) {
           G.cvs[j] += Fj.coeffs[i]/n;
         }
       }
25     return G;
     }
```

Lines 2–10 implement Step 1, inserting knots into `F`'s knot sequence. Line 11 implements Step 2, creating the B-spline `G` with the right knot sequence. Lines 15–18 implement Step 3(a), deleting a knot from `F` if the knot is different from the previous knot; otherwise, it reuses the B-spline `Fj` from the last iteration. Lines 20–23 implement step 3(b), adding the value of the control points of `Fj` to those of `G`.

Evaluation

The code is very short, and is a direct translation from the analysis.

The best case for run time occurs when all knots have full multiplicity. Then, no knots need to be inserted or deleted, and the algorithm simply becomes a less efficient version of Bézier degree raising: the algorithm uses $n(L + 2n - 1)$ additions of points to compute the P_i.

The worst case occurs when each knot has multiplicity one. In this case, increasing the multiplicity of each breakpoint requires inserting $L/2$ knots, and the cost of each knot insertion is n combinations. Deleting a knot also costs n combinations, and $L/2 + 2n - 2$ knots must be deleted. Thus, in total, the algorithm runs in $O(nL + n^2)$.

In comparison, the naive method of computing each blossom value of Equation 6.2 individually requires n blossom values for each of $n + L$ new control point, for a total of $nL + n^2$ full evaluations. A full evaluation requires $n(n-1)/2$ combinations, so the naive method runs in $O(n^3L + n^4)$. The other approach is to compute each blossom value only once. This approach requires $O(n^2L + n^3)$ combinations, since on average each blossom value is used for n consecutive control points. Thus, the new algorithm saves a factor of n or n^2 in the run time.

The algorithm uses the control points resulting from earlier knot insertions for later knot insertions. Thus, a concern is that round-off errors may accumulate so that by the time the last knot is inserted, the result is no longer accurate. Fortunately, round-off errors do not accumulate excessively. To see this is the case, observe what happens as knots are added. To increase the multiplicity of knot t_q, the knot insertion algorithm (Boehm's algorithm) computes the new control point as follows [1]:

$$Q_j \longleftarrow \frac{t_{j+n} - t_q}{t_{j+n} - t_j} Q_{j-1} + \frac{t_q - t_j}{t_{j+n} - t_j} Q_j, \quad j = q - n + 1, \ldots, q.$$

The next step would be to insert knot t_{q+2} (assuming multiplicities of 1). Thus, a control point gets modified by at most $\lceil n/2 \rceil$ successive knot insertions. In other words, it has a depth of $n/2$, half the depth of a full evaluation.

6.6 Chapter Summary

This chapter showed that simple, commonly performed tasks are simple to implement with the library. The library works well with other datatypes and routines, for example, user-created datatypes for B-spline blossoms, and STL routines. The library's datatypes and operations are general, and can be used to code non-trivial algorithms, such as basis conversion, polynomial composition and B-spline degree raising. The datatypes and operations allow direct translation from blossoming analysis to C++ code, and the resulting code is simple and meaningful in terms of blossoming analysis.

Chapter 7

Conclusions

The Blossom Classes library is a general programming tool that is useful for prototyping CAGD modeling techniques. Its main features are that it is general and it is useful for implementing many different techniques.

It provides blossoming datatypes that simplify translation from analysis to code. The datatypes and operations are general and easy to use. They allow direct translation from blossoming analysis to C++ code. The resulting code is simple and meaningful in terms of blossoming analysis.

The datatypes handle many techniques, including important ones such as Bézier curves and surfaces, and B-splines curves, and rational polynomial versions of these techniques, among others.

The operations are computed by efficient algorithms that are generalizations of important algorithms for B-splines and Bézier curves and surfaces. They can handle non-trivial algorithms.

The library works well with other libraries and datatypes. More importantly, users can replace the library's own datatypes with user-supplied datatypes. This ability to integrate user's datatypes into the library is important for two reasons: it means users can fit the library into existing applications, and users can specialize the datatypes for greater efficiency or extend them for greater functionality.

Thus, the library is suitable for many situations. The library provides datatypes that handle common modeling operations for simple applications. On the other hand, the library can be used with other tools, such as optimization routines, to create complex modeling techniques.

Future Work

The library currently only supports B-bases directly. It is desirable to extend the basic framework of the library to allow other bases to be used. Some work has been done that relates B-bases to other bases, such as convolution bases, Pólya bases, and L-bases [14, 13].

A datatype for geometrically continuous curves (universal splines [18]) should be supported, in the same way as the B-spline datatype in Chapter 6.

A datatype for tensor-product surfaces need to be supported in order for the library to be generally useful. They can be implemented in a way similar to the B-spline class.

Finally, while a simple relationship is known between a B-spline and the blossoms of its segments, no similar relationship is known for a multivariate B-spline [6] and its individual patches; currently it is known that the multivariate B-spline agrees with a B-patch in a special region. If such a relationship is found, the library should be extended to support a multivariate B-spline datatype. I hope that the functionality provided by this library can assist in such research.

Appendix A

Datatype Requirements

The following description of requirements for datatypes use the same terminology as STL. These requirement are given as a set of valid expressions in C++. Any operation that modifies its parameters are noted.

In addition to the operation listed in the table, all types must provide default constructors, copy constructors, and assignment operators.

A.1 Geometry Datatypes

A.1.1 Requirements for Scalars

In the following table, `s` and `t` are `scalars`.

expression	return type	notes
• `s+t`	`scalar`	
• `s+=t`		
• `s*t`	`scalar`	
• `s*=t`		
• `scalar(0)`	`scalar`	constructs zero scalar
• `scalar(1)`	`scalar`	constructs one scalar

A.1.2 Requirements for Ranges

In the following table,

- `sbegin` and `send` are iterators returning `scalars`
- `sbegin` is an iterator returning `rangept`.

expression	return type	notes
• `combination(sbegin, send, pbegin, (rangept*)0)`	`rangept`	returns the (affine or linear) combination of the scalars in `sbegin` to `send` with the points in `pbegin`.

A.1.3 Requirements for Domains

In the following table,

- `x` is a `space` object
- `b` is a `basis` object
- `c` is a `coordarray`
- `p` is a (domain) `point`
- `k` is an `int`
- `pbegin` is a forward iterator returning `points`

expression	return type	notes
• `domain::space`	`space`	
• `domain::basis`	`basis`	
• `domain::point`	`point`	
• `domain::coords`	`coords`	
• `domain::scalar`	`scalar`	
• `getDimension(x)`	`int`	
• `getStdBasis(x)`	`basis`	
• `getCoordsFromBasis(b,p,c)`		`c` is modified
• `getSpace(x)`	`space`	
• `getSpace(b)`	`space`	
• `getBasisElement(b,k)`	`point`	
• `setBasisElement(b,k,p)`		`b` is modified
• `makeBasis(pbegin,(basis*)0)`	`basis`	
• `coordarray(k,0)`	`coordarray`	construct an array of size `k`, initialized with zeros
• `c[k]`	reference to `scalar`	the `k`th coordinate in the array.

A.2 Blossoming Datatypes

A.2.1 Requirements for Blossoms

In the following table,

- `f` is a `blossom`

- `kn` is a `knotnet` object
- `a` is a `triarray` object (storing objects of type `blossom::rangept`)

expression	return type	notes
• `blossom::knotnet`	knotnet	
• `blossom::triarray`	triarray	
• `blossom::domain`	domain	
• `blossom::rangept`	rangept	
• `getDegree(f)`	int	
• `getSpace(f)`	space	
• `f.coeffArray()`	reference to `triarray`	
• `f.knotNet()`	reference to `knotnet`	
• `makeBlossom(kn,a,(blossom*)0)`	`blossom`	

A.2.2 Requirements for Triangular Arrays

A `triarray` satisfies all the requirements of an STL *container*. The following table lists additional requirement of a `triarray`.

In the following table,

- `a` is a `triarray` object
- `k` is an `int`
- `e` is an `evalordarray`

expression	return type	notes
• `triarray::multiindex`	multiindex	
• `triarray::evalordarray`	evalordarray	
• `a[k]`	`triarray::reference`	return the element at the position `k`. The return type is `triarray::const_reference` if `a` is constant.
• `getDimension(a)` `getDegree(a)`	`int`	dimension and degree of `a`
• `evalordarray(k,0)`	`evalordarray`	construct an array of size `k`, initialized with zeros
• `e[k]`	reference to `int`	the `k`th position in the evaluation subarray.

A.2.3 Requirements for Multi-Indices

In the following table,

- `i` is a `multiindex` object

- `k`, `l`, `d`, `n` are `int`s
- `e` is an `evalordarray`

expression	return type	notes
• `i[k]`	`int`	`k`th component of multi-index
• `ord(i,d,n)`	`int`	position of multi-index `i` in `triarray` of dimension `d`, degree `n`
• `evalord(i,d,n,e)`		compute positions of `i`th evaluation subarray in `triarray` of dimension `d`, degree `n`. `e` is modified.
• `firstTriIndex(d,n)` `lastTriIndex(d,n)`	`multiindex`	first and last indices for dimension `d`, degree `n`
• `nextTriIndex(i)` `prevTriIndex(i)`		increment/decrement index. `i` is modified.
• `isFirstIndex(i,d,n)` `isLastIndex(i,d,n)`	`int`	check if is first or last index for dimension `d`, degree `n`
• `firstSliceIndex(k,l,d,n)` `lastSliceIndex(k,l,d,n)`	`multiindex`	first and last indices for dimension `d`, degree `n` slice such that `i[k]==l`
• `nextSliceIndex(i,k,l)` `prevSliceIndex(i,k,l)`		increment/decrement index while keeping `i[k]=l`. `i` is modified
• `isFirstSliceIndex(i,k,l,d,n)` `isLastSliceIndex(i,k,l,d,n)`	int	check if is first or last index for dimension `d`, degree `n` slice such that `i[k]==l`

A.2.4 Requirements for Knot Nets

In the following table,

- `kn` is a `knot net` object
- `i` is a `multiindex` object
- `k`, `l` are `int`s
- `c` is a `combiner`
- `w` is a `weigher`
- `coeffs` is an iterator returning `rangept`
- `coords` is an iterator returning `scalar`
- `s` is a `scalar`
- `p` is a domain `point`
- `b` is a `basis`

expression	return type	notes
• `knotnet::combiner`	`combiner`	
• `knotnet::weigher`	`weigher`	
• `getCombiner(kn,p)`	`combiner`	
• `getWeigher(kn,p)`	`weigher`	
• `combineCoeffs(c,i,coeffs)`	`rangept`	the combine-coefficients algorithm.
• `weighCoords(w,i,s,coords)`		the weigh-coordinates algorithm.
• `getKnot(kn,k,l)`	`point`	
• `getDomainBasis(kn)`	`basis`	$\{t_{0,0}, \ldots, t_{d,0}\}$.
• `getDomainElement(kn,k)`	`basis`	`k`th domain element, $t_{k,0}$.
• `getEvalBasis(kn,i)`	`basis`	return the `i`th evaluation basis.
• `setKnot(kn,k,l,p)`		`kn` is modified. Bézier knot nets do not support this operation.
• `swapKnot(kn,k,l)`		swap the knots `(k,l)` and `(k,l+1)`. `kn` is modified. Bézier knot nets do not support this operation.
• `setDomainBasis(kn,b)`	`basis`	sets all domain elements to basis `b`'s elements.
• `setDomainElement(kn,k,p)`	`basis`	sets all knots `(k,l)` to `p`.

Bibliography

[1] Phillip J. Barry and Ronald N. Goldman. Knot insertion algorithms via blossoming. In Hans-Peter Seidel, editor, *Blossoming: The new polar-form approach to spline curves and surfaces*. SIGGRAPH Course Notes #23, 1991.

[2] R. Bartels. Object oriented spline software. In Pierre-Jean Laurent, Alain Le Méhauté, and Larry L. Schumaker, editors, *Curves and Surfaces in gemetric design*, pages 27–34. A K Peters Ltd, 1994.

[3] W. Boehm. Inserting new knots into B-spline curves. *Computer-Aided Design*, 12:199–201, 1980.

[4] W. Boehm and H. Prautzsch. The insertion algorithm. *Computer-Aided Design*, 17:58–59, 1985.

[5] A. Dahl. Weyl: A language for computer graphics and computer aided geometric design. Technical Report TR 92-06-02, University of Washington, June 1992.

[6] W. Dahmen, C. A. Micchelli, and H. P. Seidel. Blossoming begets B-splines built better by B-patches. *Mathematics of Computation*, 59:297–320, 1992.

[7] Paul de Faget de Casteljau. *Formes à Pôles*, volume 2 of *Mathématiques et CAO*. Hermes, 51 rue Rennequin, 75017 Paris, 1985.

[8] Derose, Goldman, Haen, and Mann. Composition algorithm via blossoming: Theory, applications and implementation. Technical Report 91-05-04, University of Washington, 1991.

[9] T. DeRose and R. Goldman. A tutorial introduction to blossoming. In H. Hagen and D. Roller, editors, *Geometric Modeling*. Springer, 1991.

[10] Tony D. DeRose. A coordinate-free approach to geometric programming. In *Math for SIGGRAPH*. SIGGRAPH Course Notes #23, 1989. Also available as Technical Report No. 89-09-16, Department of Computer Science and Engineering, University of Washington, Seattle, WA (September, 1989).

[11] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, third edition, 1990.

[12] Goldman and Barry. Wonderful triangle: A simple, unified algorithmic approach to change of basis procedures in CAGD. In Tom Lyche and Larry L. Schumaker, editors, *Mathematical Methods in CAGD II*. Academic Press, Inc, 1992.

[13] R. N. Goldman. Recursive triangles. In Wolfgang Dahmen, Mariano Gasca, and Charles A. Micchelli, editors, *Computation of curves and surfaces*, volume 307 of *NATO ASI Series C: Mathematical and Physical Sciences*, pages 27–72. Kluwer Academic Publishers, 1990.

[14] S. Lodha and R. Goldman. A multi-variate de Boor-fix formula. In Pierre-Jean Laurent, Alain Le Méhauté, and Larry L. Schumaker, editors, *Curves and Surfaces in Geometric Design*, pages 301–310. A K Peters Ltd, 1994.

[15] Stephen Mann and Wayne Liu. An analysis of polynomial composition algorithms. Technical Report CS-95-24, University of Waterloo, Waterloo, Ontario, N2L 3G1 CANADA, 1995.

[16] Lyle Ramshaw. Blossoming: A connect-the-dots approach to splines. Technical Report 19, Digital Equipment Corporation, Systems Research Centre, 21 June 1987.

[17] H. P. Seidel. Symmetric recursive algorithms for surfaces: B-patches and the de Boor algorithm for polynomials over triangles. *Constructive Approximation*, 7:259–279, 1991.

[18] Hans-Peter Seidel. Polar forms for geometrically continuous spline curves of arbitrary degree. *ACM Transcations on Graphics*, 12(1), January 1993.

[19] K. Shoemake. Animating rotation with quaternion curves. In *Computer Graphics (Proc. of SIGGRAPH '85)*, pages 245–254, 1985.

[20] Ken Shoemake. Efficient de Casteljau indexing. In preparation, 1995.

[21] Alexander Stepanov and Meng Lee. The standard template library. Technical Report HPL-94-34, Hewlett-Packard Labroatories, April 1994.