B-Spline Interpolation on Lattices

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: July 15, 1999
Last Modified: September 11, 2020

Contents

1 Introduction

This document describes B-spline interpolation of data organized as uniformly spaced samples on a lattice in multiple dimensions. The interpolation is useful for generating continuous representations of multidimensional images. The terminology is that used in [1]. An implementation is provided by IntpBSplineUniform.h. It allows for dimensions known at compile time and for dimensions known only at run time. The code also has specializations for dimensions 1, 2 and 3. A sample application is in the GTE/Samples/Imagics/BSplineInterpolation folder.

2 B-Spline Basis Functions

A *knot vector* is a set of $n+1$ real numbers $\mathbf{t} = \{t_i\}_{i=0}^{n}$, where $t_i \le t_{i+1}$ for all i. It is possible that some of the elements are equal. The set of unique elements is called a *breakpoint sequence*, say, $\mathbf{u} = \{u_i\}_{i=0}^{s}$, a set of $s+1$ real numbers with $u_i < u_{i+1}$ for all i. The *multiplicity vector* associated with $\mathbf{u}$ is $\mathbf{m} = \{m_i\}_{i=0}^{s}$, a set of positive integers. Define $\rho_0 = 0$ and $\rho_j = \sum_{i=0}^{j-1} m_j$ for $1 \le j \le s+1$, in which case $\rho_{s+1} = n+1$. The knots satisfy the conditions $t_k = u_j$ for $\rho_j \le k < \rho_{j+1}$ and for $0 \le j \le s$.

The *control points* for a curve are $\{\mathbf{F}_i\}_{i=0}^{c}$, a set of $c+1$ points in $\mathbb{R}^\alpha$ where $\alpha \ge 1$. We can construct a piecewise polynomial curve $\mathbf{X}(t)$ of degree $d \ge 1$ from the control points and a knot vector, say

$$\mathbf{X}(t) = \sum_{i=0}^{c} \mathbf{F}_i B_{i,d}(t) \tag{1}$$

where $B_{i,d}(t)$ is a *B-spline basis function* defined by a recursive algorithm involving the knots. The algorithm requires that the number of control points, the number of knots and the degree of the polynomial pieces are related by $n+1 = (c+1)+d+1$. The curve is referred to as a *B-spline curve*.

The B-spline basis functions are defined as follows. Define

$$B_{i,0}(t) = \begin{cases} 1, & t \in [t_i, t_{i+1}) \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

for $0 \le i \le n-1$. Recursively define

$$B_{i,j}(t) = \begin{cases} \left(\frac{t-t_i}{t_{i+j}-t_i}\right) B_{i,j-1}(t) + \left(\frac{t_{i+1+j}-t}{t_{i+1+j}-t_{i+1}}\right) B_{i+1,j-1}(t), & t \in [t_i, t_{i+1+j}) \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

for $1 \le j \le d$ and $0 \le i \le n-1-j$. Observe that if we have repeated knots, the denominator $t_{i+j} - t_i$ is zero for those j involving the repetitions. The division by zero is not allowed, but it turns out that when the denominator is zero, $B_{i,j-1}(t)$ is also identically zero. The same situation can occur for the other term when $t_{i+1+j} - t_{i+1}$ is zero, in which case $B_{i+1,j-1}(t)$ is identically zero. The convention is that when a basis function on the right-hand side is zero, its coefficient is not evaluated and the division by zero does not occur.

The domain of the B-spline curve is $\mathbb{R}$, but of greater interest is the *support* of the curve. Generally, the support of a function $f(t)$ is the smallest interval I for which $f(t) = 0$ for all $t \notin I$. The support of the B-spline curve is $[t_0, t_n]$.

The B-spline curve of equation (1) is said to be an *open curve* when $t_0 = u_0$ has multiplicity $m_0 \geq d+1$ and $t_n = u_s$ has multiplicity $m_s \geq d+1$. For example, consider a curve of degree $d = 2$ with $c+1 = 7$ control points. The number of required knots is $n+1 = 10$. The knots $\mathbf{t} = \{0, 0, 0, 1, 2, 4, 8, 9, 9, 9\}$ have $s+1 = 6$ unique values and generate an open curve because $t_0 = t_1 = t_2$ (the multiplicity is $m_0 = 3 \geq d+1$) and $t_7 = t_8 = t_9$ (the multiplicity is $m_5 = 3 \geq d+1$). Such a curve interpolates $\mathbf{F}_0$ and $\mathbf{F}_c$; that is, the curve passes through these points. The curve has a tangent in the direction $\mathbf{F}_1 - \mathbf{F}_0$ at t_d and a tangent in the direction $\mathbf{F}_c - \mathbf{F}_{c-1}$ at t_{n-d}. Additionally, the curve is said to be open and *uniform* when $t_{i+1} - t_i = \Delta$, a constant, for $d \leq i \leq n-d-1$.

If a B-spline curve is not open, it is said to be *floating*[1]. Such curves generally do not interpolate the end control points—the curve "floats" near them. The fullset of knots for a floating curve can be equally spaced, in which case the curve is said to be floating and uniform. The knot differences are $t_{i+1} - t_i = \Delta$, a constant, for $0 \leq t \leq n-1$.

Example 1 shows several B-spline basis functions for a set of open uniform knots. The generation of the functions and the graphical display were created by [2].

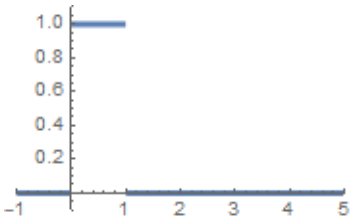
Example 1. *B-spline basis functions for a set of open uniform knots.* Let the knot vector be $\mathbf{t} = \{0, 0, 0, 1, 2, 3, 4, 4, 4\}$, so $n+1 = 9$. Let the degree be $d = 2$. A B-spline curve using these knots and having the specified degree must have $c+1 = (n+1) - (d+1) = 6$ control points.

The basis functions $B_{0,0}(t)$, $B_{1,0}(t)$, $B_{6,0}(t)$ and $B_{7,0}(t)$ are all identically zero. The other basis functions for $j = 0$ and $2 \leq i \leq 5$ are

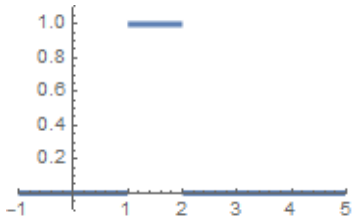
$$\begin{aligned} B_{2,0}(t) &= \{1,\ t \in [0,1);\ 0,\ \text{otherwise}\} \\ B_{3,0}(t) &= \{1,\ t \in [1,2);\ 0,\ \text{otherwise}\} \\ B_{4,0}(t) &= \{1,\ t \in [2,3);\ 0,\ \text{otherwise}\} \\ B_{5,0}(t) &= \{1,\ t \in [3,4);\ 0,\ \text{otherwise}\} \end{aligned}$$

The graphs of $B_{i,0}(t)$ for $2 \leq i \leq 5$ are shown next.

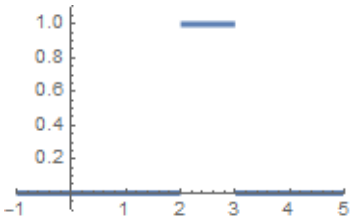
[1] For now I am ignoring the distinction between *periodic* and *aperiodic* (open, floating) curves.



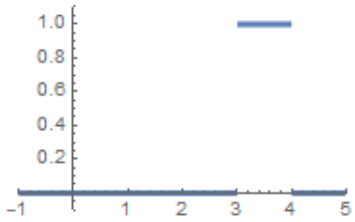
graph of $B_{2,0}(t)$



graph of $B_{3,0}(t)$



graph of $B_{4,0}(t)$

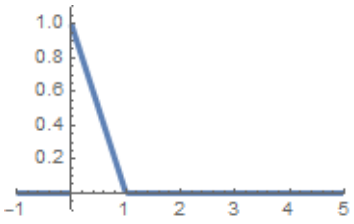


graph of $B_{5,0}(t)$

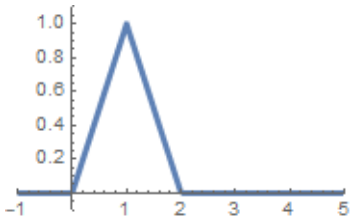
The basis functions $B_{0,1}(t)$ and $B_{6,1}(t)$ are identically zero. The other basis functions for $j = 1$ and $1 \le i \le 5$ are

$$
\begin{aligned}
B_{1,1}(t) &= \{1-t,\ t \in [0,1);\ 0,\ \text{otherwise}\} \\
B_{2,1}(t) &= \{t,\ t \in [0,1);\ 2-t,\ t \in [1,2);\ 0,\ \text{otherwise}\} \\
B_{3,1}(t) &= \{-1+t,\ t \in [1,2);\ 3-t,\ t \in [2,3);\ 0,\ \text{otherwise}\} \\
B_{4,1}(t) &= \{-2+t,\ t \in [2,3);\ 4-t,\ t \in [3,4);\ 0,\ \text{otherwise}\} \\
B_{5,1}(t) &= \{-3+t,\ t \in [3,4);\ 0,\ \text{otherwise}\}
\end{aligned}
$$

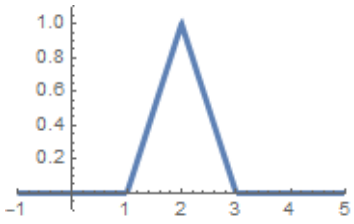
The graphs of $B_{i,1}(t)$ for $1 \le i \le 5$ are shown next.



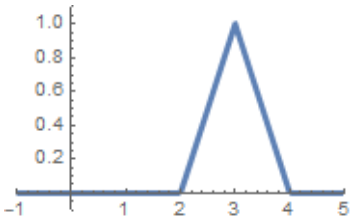
graph of $B_{1,1}(t)$



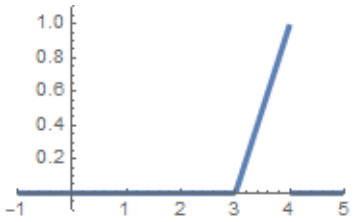
graph of $B_{2,1}(t)$



graph of $B_{3,1}(t)$



graph of $B_{4,1}(t)$

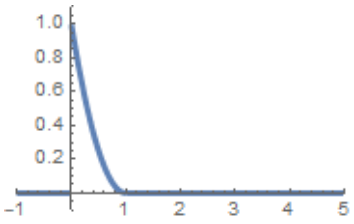


graph of $B_{5,1}(t)$

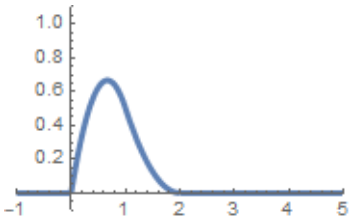
The basis functions for $j = 2$ and $0 \le i \le 5$ are

$$
\begin{aligned}
B_{0,2}(t) &= \left\{(1-t)^2,\ t \in [0,1);\ 0,\ \text{otherwise}\right\} \\
B_{1,2}(t) &= \left\{2t - 3t^2/2,\ t \in [0,1);\ (2-t)^2/2,\ t \in [1,2);\ 0,\ \text{otherwise}\right\} \\
B_{2,2}(t) &= \left\{t^2/2,\ t \in [0,1);\ (-3+6t-2t^2)/2,\ t \in [1,2);\ (3-t)^2/2,\ t \in [2,3);\ 0,\ \text{otherwise}\right\} \\
B_{3,2}(t) &= \left\{(1-t)^2/2,\ t \in [1,2);\ (-11+10t-2t^2)/2,\ t \in [2,3);\ (4-t)^2/2,\ t \in [3,4);\ 0,\ \text{otherwise}\right\} \\
B_{4,2}(t) &= \left\{(2-t)^2/2,\ t \in [2,3);\ (-20+13t-2t^2)/2,\ t \in [3,4);\ 0,\ \text{otherwise}\right\} \\
B_{5,2}(t) &= \left\{(3-t)^2,\ t \in [3,4);\ 0,\ \text{otherwise}\right\}
\end{aligned}
$$

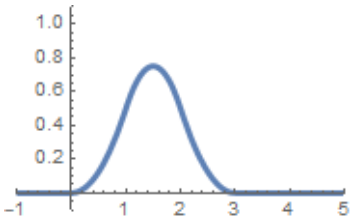
The graphs of $B_{i,2}(t)$ for $0 \le i \le 5$ are shown next.



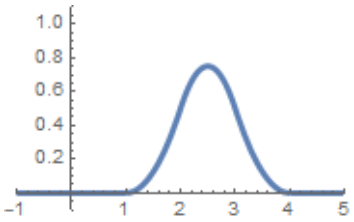
graph of $B_{0,2}(t)$



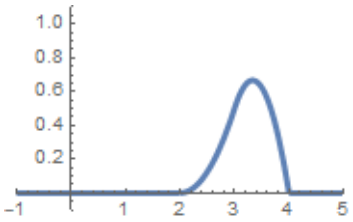
graph of $B_{1,2}(t)$



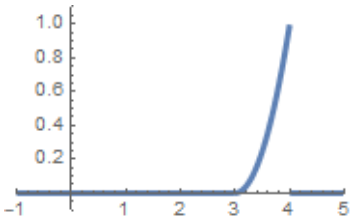
graph of $B_{2,2}(t)$



graph of $B_{3,2}(t)$

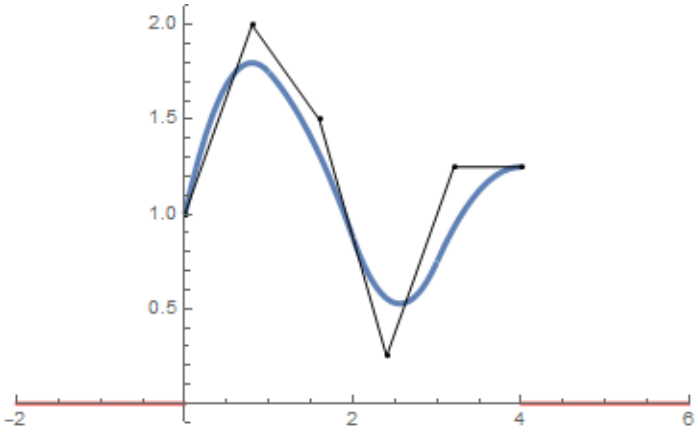


graph of $B_{4,2}(t)$



graph of $B_{5,2}(t)$

The support of the B-spline curve is $[t_2, t_6] = [0, 4]$. Choose control points $F_0 = 1$, $F_1 = 2$, $F_2 = 3/2$, $F_3 = 1/4$, $F_4 = 5/4$ and $F_5 = 5/4$. Assume these occur at equally spaced t-values on $[0, 4]$, specifically, $\{0, 4/5, 8/5, 12/5, 16/5, 4\}$. The graph of the B-spline curve $\sum_{i=0}^{5} F_i B_{i,2}(t)$ is shown next.



The control points are drawn in black and connected by a polyline drawn in black. The B-spline curve is drawn in blue. In the example, the support is $[t_2, t_6] = [0, 4]$. The graph shows zero-valued curves in the intervals $[-2, 0]$ and $[4, 6]$, drawn in red, and is used for comparison to the floating uniform curve discussed in the next example.

Example 2 shows several B-spline basis functions for a set of floating uniform knots. The generation of the functions and the graphical display were created by [2].

Example 2. *B-spline basis functions for a set of floating uniform knots.* Let the knot vector be $\mathbf{t} = \{-2, -1, 0, 1, 2, 3, 4, 5, 6\}$, so $n + 1 = 9$. Let the degree be $d = 2$. A B-spline curve using these knots and having the specified degree must have $c + 1 = (n + 1) - (d + 1) = 6$ control points.

The basis functions $B_{i,0}(t)$ are all unit step functions,

$$B_{i,0}(t) = \begin{cases} 1, & t \in [i-2, i-1) \\ 0, & \text{otherwise} \end{cases}$$

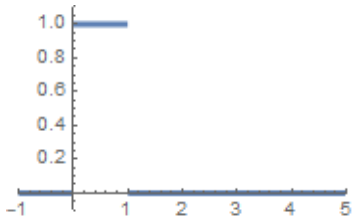
for $0 \le i \le 7$. The basis functions $B_{i,1}(t)$ are translations of the same function,

$$B_{i,1}(t) = \begin{cases} t + 2 - i, & t \in [i-2, i-1) \\ i - t, & t \in [i-1, i) \\ 0, & \text{otherwise} \end{cases}$$

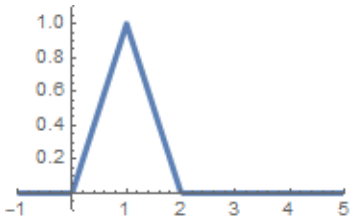
for $0 \le i \le 6$. The basis functions $B_{i,2}(t)$ are translations of the same function,

$$B_{i,2}(t) = \begin{cases} (t+2-i)^2/2, & t \in [i-2, i-1) \\ ((t+2-i)(i-t) + (i+1-t)(t-i+1))/2, & t \in [i-1, i) \\ (t-i-1)^2/2, & t \in [i, i+1) \\ 0, & \text{otherwise} \end{cases}$$

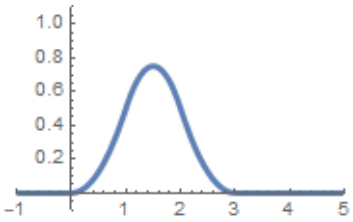
for $0 \le i \le 5$. The graphs of the functions whose translations form the basis functions are shown next.



graph of $B_{2,0}(t)$

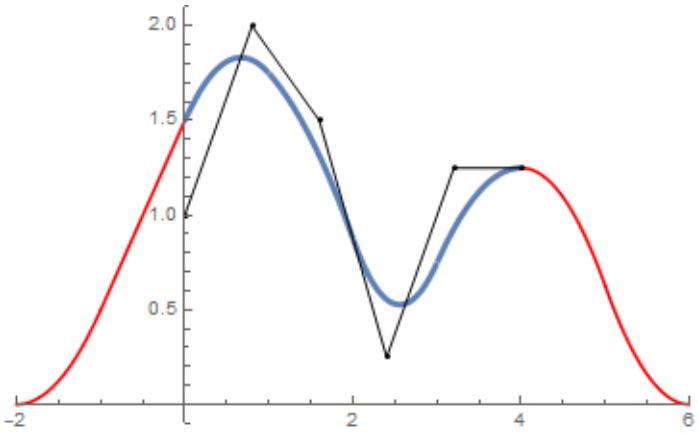


graph of $B_{2,1}(t)$



graph of $B_{2,2}(t)$

The support of the B-spline curve is $[t_0, t_8] = [-2, 6]$, but for comparison to the open B-spline curve, we care only about the subinterval $[0, 4]$. Choose control points $F_0 = 1$, $F_1 = 2$, $F_2 = 3/2$, $F_3 = 1/4$, $F_4 = 5/4$ and $F_5 = 5/4$. Assume these occur at equally spaced t-values on $[0, 4]$, $\{0, 4/5, 8/5, 12/5, 16/5, 4\}$. The graph of the B-spline curve $\sum_{i=0}^{5} F_i B_{i,2}(t)$ is shown next,



The control points are drawn in black and connected by a polyline drawn in black. The B-spline curve is drawn in blue for the domain $[0, 4]$ that is common with the open B-spline curve. The graph shows the remaining portions of the curve for the t-intervals $[-2, 0]$ and $[4, 6]$, drawn in red. As expected, the height values at $t_0 = -2$ and $t_8 = 6$ are zero the support is $[-2, 6]$. Compare this graph to the one shown for the open uniform curve discussed in the previous example.

---

3 Floating Uniform Knots

The B-spline basis functions with floating uniform knots are used for B-spline interpolation of image data. Let us analyze these functions in more depth. As shown in equations (2) and (3), the basis function $B_{i,j}(t)$ has support $[t_i, t_{i+1+j}]$. I will refer to the half-closed interval $[t_i, t_{i+1+j})$ as the t-domain of the function; later we will have a change of variables introducting an s-variable and need to distinguish between s and t and between support and domain. As example 2 illustrates, $B_{i,j}(t)$ is a piecewise polynomial function on its support. $B_{i,0}(t)$ is piecewise constant, $B_{i,1}(t)$ is piecewise linear, $B_{i,2}(t)$ is piecewise quadratic, and so on. Generally, $B_{i,j}(t)$ is piecewise polynomial with $j + 1$ polynomials each having degree j. The goal is to construct formulas for the polynomials.

3.1 Reduction to Integer-Valued Knots

It is sufficient to analyze the basis functions using a change of variables that converts the constant spacing between knots to unit length. Let $\Delta > 0$ be the constant spacing; that is, $t_{i+1} - t_i = \Delta$ for all i. Given a starting knot t_0, the knots are $t_i = t_0 + i\Delta$ for $0 \le i \le n$. Consider the change of variables $t = t_0 + s\Delta$. Define the s-knots by $s_i = (t_i - t_0)/\Delta$, in which case $s_{i+1} - s_i = 1$ for all i. This leads to $s_i = i$ for all i because $s_0 = 0$. The recursive formulas for the B-spline basis functions may be written in terms of the s-variable,

$$B_{i,0}(s) = \begin{cases} 1, & s \in [i, i+1) \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

for $0 \le i \le n-1$ and

$$B_{i,j}(s) = \begin{cases} \left(\frac{s-i}{j}\right) B_{i,j-1}(s) + \left(\frac{i+1+j-s}{j}\right) B_{i+1,j-1}(s), & s \in [i, i+1+j) \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

for $1 \le j \le d$ and $0 \le i \le n-1-j$. As observed previously, the basis functions for a specified j are translations of a single function, so $B_{i+\ell,j}(s) = B_{i,j}(s-\ell)$. The single functions are indexed solely by j, say, $\hat{B}_j(s) = B_{0,j}(s)$. The implication is that $B_{i,j}(s) = \hat{B}_j(s-i)$.

3.2 B-Spline Curve for Integer Knots

Using the construction in the previous section, the B-spline curve of degree d generated by the basis functions is

$$\mathbf{X}(s) = \sum_{i=0}^{c} \mathbf{F}_i B_{i,d}(s) = \sum_{i=0}^{c} \mathbf{F}_i \hat{B}_d(s-i) \tag{6}$$

Equation (6) is a convolution of the control points with the basis function $\hat{B}_d(s)$. The basis function has support $[0, d+1]$ and has s-domain $[d, n-d) = [d, c+1)$. On this domain we are guaranteed that for any selected s, $\mathbf{X}(s)$ involves a sum of $d+1$ terms with the remaining terms zero because s is outside the domains of the corresponding basis functions. To illustrate, the t-domain of the B-spline curve in example 2 is $[0, 4]$. The change of variables from t to s is $t = s-2$, so the s-domain of the curve is $[2, 6]$. The B-spline curve evaluation is shown in Table 1, including the boundary evaluations for s outside the domain,

Table 1. Evaluation of the B-spline curve shows the local control provided by the B-spline basis functions.

s-interval	$\mathbf{X}$-curve	evaluation
$(-\infty, 0)$		0
$[0, 1)$	$\mathbf{X}_0(s)$	$F_0\hat{B}_2(s)$
$[1, 2)$	$\mathbf{X}_1(s)$	$F_0\hat{B}_2(s) + F_1\hat{B}_2(s-1)$
$[2, 3)$	$\mathbf{X}_2(s)$	$F_0\hat{B}_2(s) + F_1\hat{B}_2(s-1) + F_2\hat{B}_2(s-2)$
$[3, 4)$	$\mathbf{X}_3(s)$	$F_1\hat{B}_2(s-1) + F_2\hat{B}_2(s-2) + F_3\hat{B}_2(s-3)$
$[4, 5)$	$\mathbf{X}_4(s)$	$F_2\hat{B}_2(s-2) + F_3\hat{B}_2(s-3) + F_4\hat{B}_2(s-4)$
$[5, 6)$	$\mathbf{X}_5(s)$	$F_3\hat{B}_2(s-3) + F_4\hat{B}_2(s-4) + F_5\hat{B}_2(s-5)$
$[6, 7)$	$\mathbf{X}_6(s)$	$F_4\hat{B}_2(s-4) + F_5\hat{B}_2(s-5)$
$[7, 8)$	$\mathbf{X}_7(s)$	$F_5\hat{B}_2(s-5)$
$[8, +\infty)$		0

The evaluations on the s-domain $[2, 6]$ are shown between the dashed lines of the table. The curve pieces $\mathbf{X}_2(s)$ through $\mathbf{X}_5(s)$ are the only ones relevant for the evaluation. Observe that the naming $\mathbf{X}_i(s)$ is suggestive that the s-domain is $[i, i+1)$.

3.3 Construction of the Basis Function Polynomial Pieces

The recursive formulas reduce to the following, essentially setting $i = 0$ in equations (4) and (5) and using the translation equivalence,

$$\hat{B}_0(s) = \begin{cases} 1, & s \in [0,1) \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

and

$$\hat{B}_j(s) = \begin{cases} \left(\frac{s}{j}\right)\hat{B}_{j-1}(s) + \left(\frac{j+1-s}{j}\right)\hat{B}_{j-1}(s-1), & s \in [0, j+1) \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

The polynomials are $\hat{B}_j(s) = P_{j,k}(s)$ for $s \in [k, k+1)$ where $0 \le k \le j$, a set of $j+1$ polynomials each with degree j. Equation (7) implies

$$P_{0,0}(s) = 1, \quad s \in [0,1) \tag{9}$$

a constant polynomial. When $j = 1$, we have

$$\hat{B}_1(s) = s\hat{B}_0(s) + (2-s)\hat{B}_0(s-1), \quad s \in [0,2) \tag{10}$$

$\hat{B}_0(s)$ is defined for $s \in [0,1)$. $\hat{B}_0(s-1)$ is defined for $s - 1 \in [0,1)$, that is, for $s \in [1,2)$. Therefore, $B_1(s) = s$ for $s \in [0,1)$ and $B_1(s) = 2 - s$ for $s \in [1,2)$. The polynomials of degree 1 in the recursion are

$$\begin{aligned} P_{1,0}(s) &= s, & s &\in [0,1) \\ P_{1,1}(s) &= -s + 2 = 1 - (s-1), & s &\in [1,2) \end{aligned} \tag{11}$$

which is consistent with the graph of $B_{2,1}(t)$ in example 2. When $j = 2$, we have

$$\hat{B}_2(s) = (s/2)\hat{B}_1(s) + ((3-s)/2)\hat{B}_1(s-1), \quad s \in [0,3) \tag{12}$$

$\hat{B}_1(s)$ is defined for $s \in [0,2)$. $\hat{B}_1(s-1)$ is defined for $s \in [1,3)$. Therefore, $B_2(s) = (s/2)s$ for $s \in [0,1)$, $B_2(s) = (s/2)(2-s) + ((3-s)/2)(s-1)$ for $s \in [1,2)$ and $B_2(s) = ((3-s)/2)(2-(s-1))$ for $s \in [1,3)$. The polynomials of degree 2 in the recursions are

$$\begin{aligned} P_{2,0}(s) &= s^2/2, & s &\in [0,1) \\ P_{2,1}(s) &= (-2s^2 + 6s - 3)/2 = (1 + 2(s-1) - 2(s-1)^2)/2, & s &\in [1,2) \\ P_{2,2}(s) &= (s^2 - 6s + 9)/2 = (1 - 2(s-2) + (s-2)^2)/2, & s &\in [2,3) \end{aligned} \tag{13}$$

which is consistent with the graph of $B_{2,2}(t)$ in example 2.

Generally,

$$P_{j,k}(s) = (s/j)P_{j-1,k}(s) + ((j+1-s)/j)P_{j-1,k-1}(s-1), \quad s \in [k, k+1) \tag{14}$$

for $0 \le k \le j$ where $P_{j-1,k}(s)$ is defined for $s \in [0,j)$ and $P_{j-1,k}(s-1)$ is defined for $s \in [1, j+1)$. There are boundary conditions, so to speak. The polynomials $P_{j,k}(s)$ that we want are those for which $0 \le j \le k$. If $k < 0$ or $k > j$, define $P_{j,k}(s) = 0$; that is, these polynomials are identically zero for all s.

For example, consider the case $j = 3$. For $k = 0$ and $s \in [0, 1)$,

$$\begin{aligned} P_{3,0}(s) &= (s/3)P_{2,0}(s) + ((4-s)/3)P_{2,-1}(s-1) \\ &= (s/3)[s^2/2] + ((4-s)/3)[0] \\ &= s^3/6 \end{aligned} \tag{15}$$

For $k = 1$ and $s \in [1, 2)$,

$$\begin{aligned} P_{3,1}(s) &= (s/3)P_{2,1}(s) + ((4-s)/3)P_{2,0}(s-1) \\ &= (s/3)[(s(2-s) + (3-s)(s-1))/2] + ((4-s)/3)[(s-1)^2/2] \\ &= (-3s^3 + 12s^2 - 12s + 4)/6 \\ &= (1 + 3(s-1) + 3(s-1)^2 - 3(s-1)^2)/6 \end{aligned} \tag{16}$$

For $k = 2$ and $s \in [2, 3)$,

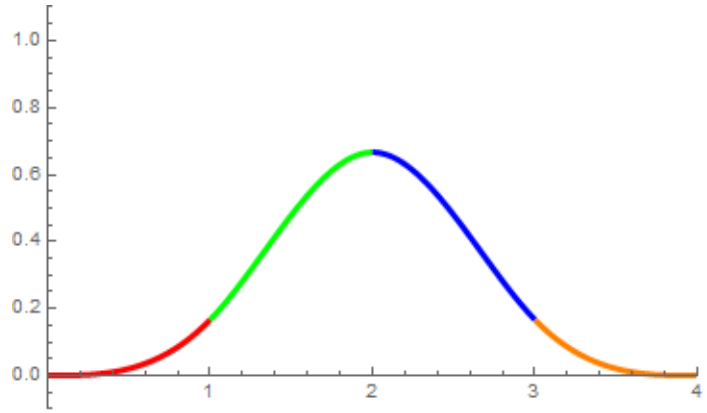
$$\begin{aligned} P_{3,2}(s) &= (s/3)P_{2,2}(s) + ((4-s)/3)P_{2,1}(s-1) \\ &= (s/3)[(3-s)^2/2] + ((4-s)/3)[((s-1)(2-(s-1)) + (3-(s-1))((s-1)-1))/2] \\ &= (3s^3 - 24s^2 + 60s - 44)/6 \\ &= (4 - 6(s-2)^2 + 3(s-2)^3)/6 \end{aligned} \tag{17}$$

For $k = 3$ and $s \in [3, 4)$,

$$\begin{aligned} P_{3,3}(s) &= (s/3)P_{2,3}(s) + ((4-s)/3)P_{2,2}(s-1) \\ &= (s/3)[0] + ((2-s)/3)[((s-1)(2-(s-1)) + (3-(s-1))((s-1)-1))/2] \\ &= (-s^3 + 12s^2 - 48s + 64)^3/6 \\ &= (1 - 3(s-3) + 3(s-3)^2 - (s-3)^3)/6 \end{aligned} \tag{18}$$

Figure 1 shows the graph of the 4 polynomial pieces.

Figure 1. The graph of the function $\hat{B}_3(s)$, composed of the graphs of $P_{3,0}(s)$ drawn in red, $P_{3,1}(s)$ drawn in green, $P_{3,2}(s)$ drawn in blue and $P_{3,3}(s)$ drawn in orange.



3.4 Construction of the B-Spline Curve Pieces

A further reduction in the structure of the B-spline curves now follows from the construction of the polynomials $P_{j,k}(s)$. In table 1 corresponding to example 2, the curves pieces are shown next. Only those curves corresponding to the s-support $[2, 6]$ are listed.

$$\begin{aligned}
X_2(s) &= F_0P_{2,2}(s-0) + F_1P_{2,1}(s-1) + F_2P_{2,0}(s-2), \quad s \in [2,3) \\
X_3(s) &= F_1P_{2,2}(s-1) + F_2P_{2,1}(s-2) + F_3P_{2,0}(s-3), \quad s \in [3,4) \\
X_4(s) &= F_2P_{2,2}(s-2) + F_3P_{2,1}(s-3) + F_4P_{2,0}(s-4), \quad s \in [4,5) \\
X_5(s) &= F_3P_{2,2}(s-3) + F_4P_{2,1}(s-4) + F_5P_{2,0}(s-5), \quad s \in [5,6)
\end{aligned} \tag{19}$$

Generally, for a B-spline curve $\mathbf{X}(s) = \sum_{i=0}^{c} \mathbf{F}_i \hat{B}_d(s-i)$ with $c+1$ control points, degree d and s-domain $[d, c+1)$, the curve pieces are

$$\mathbf{X}_{d+i}(s) = \sum_{j=0}^{d} \mathbf{F}_{i+j} P_{d,d-j}(s-i-j) \tag{20}$$

for $0 \le i \le c-d$.

Define the polynomials $Q_{d,k}(s) = P_{d,k}(s+k)$ for $0 \le k \le d$; then $P_{d,d-j}(s-i-j) = P_{d,d-j}(s+(d-j)-d-i) = Q_{d,d-j}(s-d-i)$ and Equation 20 becomes

$$\mathbf{X}_{d+i}(s) = \sum_{j=0}^{d} \mathbf{F}_{i+j} Q_{d,d-j}(s-d-i) \tag{21}$$

for $0 \le i \le c-d$ and $s \in [d+i, d+i+1)$. The translation in the Q-polynomials is independent of j. In this form, we can factor out a *blending matrix* $A_d = [a_{j\ell}^{(d)}]$ that has size $(d+1) \times (d+1)$ and combines powers of $(s-d-i)$ into the Q-polynomial terms. Equation (21) becomes

$$\mathbf{X}_{d+i}(s) = \sum_{j=0}^{d} \mathbf{F}_{i+j} \sum_{k=0}^{d} a_{jk}^{(d)} (s-d-i)^k \tag{22}$$

for $0 \le i \le c-d$ and $s \in [d+i, d+i+1)$. The blending matrices for small values of d are shown next,

$$A_1 = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \quad A_2 = \frac{1}{2}\begin{bmatrix} 1 & -2 & 1 \\ 1 & 2 & -2 \\ 0 & 0 & 1 \end{bmatrix}, \quad A_3 = \frac{1}{6}\begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_4 = \frac{1}{24}\begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ 11 & -12 & -6 & 12 & -4 \\ 11 & 12 & -6 & -12 & 6 \\ 1 & 4 & 6 & 4 & -4 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad A_5 = \frac{1}{120}\begin{bmatrix} 1 & -5 & 10 & -10 & 5 & -1 \\ 26 & -50 & 20 & 20 & -20 & 5 \\ 66 & 0 & -60 & 0 & 30 & -10 \\ 26 & 50 & 20 & -20 & -20 & 10 \\ 1 & 5 & 10 & 10 & 5 & -5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{23}$$

An implementation of B-splines can compute the matrices using code that implements univariate polynomial arithmetic. Equation (14) is used to generate the polynomials $P_{d,k}(s)$ for $0 \le k \le d$. The Q-polynomials are then generated by $Q_{d,k}(s) = P_{d,k}(s+k)$ for $0 \le k \le d$. The elements of A_d can be computed from the coefficients of the Q-polynomials. Listing 1 shows pseudocode for the computations.

Listing 1. Computation of the blending matrix A_d from the coefficients of the Q-polynomials.

```
// Input:  degree d > 0
// Output:  (d+1)x(d+1) blending matrix A
void ComputeBlendingMatrix(int d, Real A[d + 1][d + 1])
{
    // Coefficients of polynomial P[i] are P[i][0..d].
    Polynomial P[d + 1];  // default degrees are 0

    // P{0,0}(s) = 1
    P[0][0] = 1;

    // L0 = s/j, the constant term is set to 0.  The linear term 1/j will be set later.
    Polynomial> L0(1);  // input is the degree 1
    L0[0] = 0;

    // L1(s) = (j + 1 - s)/j
    Polynomial L1(1);

    // s-1 is used in computing translated P{j-1,k-1}(s-1)
    Polynomial sm1 = { -1, 1 };  // { sm1[0], sm1[1] }

    // Compute P{j,k}(s) = L0(s)*P{j-1,k}(s) + L1(s)*P{j-1,k-1}(s-1) for
    // 0 <= k <= j where 1 <= j <= degree.  When k = 0, P{j-1,-1}(s) = 0,
    // so P{j,0}(s) = L0(s)*P_{j-1,0}(s).  When k = j, P{j-1,j}(s) = 0,
    // so P{j,j}(s) = L1(s)*P_{j-1,j-1}(s).  The polynomials at level j-1
    // are currently stored in P[0] through P[j-1].  The polynomials at
    // level j are computed and stored in P[0] through P[j]; that is, they
    // are computed in place to reduce memory usage and copying.  This
    // requires computing P[k] (level j) from P[k] (level j-1) and P[k-1]
    // (level j-1), which means we have to process k = j down to k = 0.
    for (int j = 1; j <= d; ++j)
    {
        Real invJ = 1.0 / j;
        L0[1] = invJ;
```

```
            L1[0] = 1.0 + invJ;
            L1[1] = -invJ;

            for (int k = j; k >= 0; --k)
            {
                Polynomial result = { 0 };  // constant polynomial 0

                if (k > 0)
                {
                    // For polynomial Q(t), Q.GetTranslation(t0) is Q(t-t0).
                    result += L1 * P[k - 1].GetTranslation(1);
                }

                if (k < j)
                {
                    result += L0 * P[k];
                }

                P[k] = result;
            }
        }

        // Compute Q{d,k}(s) = P{d,k}(s + k).  For polynomial P(s),
        // P.GetTranslation(-k) is P(s+k).
        Polynomial Q[d + 1];
        for (int k = 0; k <= d; ++k)
        {
            Q[k] = P[k].GetTranslation(-k);
        }

        // Extract the matrix A from the Q-polynomials.  Row r of A contains the
        // coefficients of Q{d,d-r}(s).
        for (int k = 0, r = d; k <= d; ++k, --r)
        {
            for (uint32_t c = 0; c <= d; ++c)
            {
                A[r][c] = Q[k][c];
            }
        }
    }
}
```

3.5 Choosing Parameters Commensurate with Control-Point Indices

The B-spline curve has $c+1$ control points $\mathbf{F}_i$ for $0 \le i \le c$. The s-support of the B-spline curve is $[d, c+1]$ and the s-knots are consecutive integers. The number of integers in $[d, c+1]$ is $c+2-d$, which is not the number of control points when $d > 1$. Therefore, it is not possible to choose integer values in $[d, c+1]$ to correspond to the control point indices.

Although it is possible to choose a B-spline curve parameter $t \in [0, c]$ and then map it to $s \in [d, c+1]$, I prefer instead to think of the sample values chosen at the center of an interval. This is similar to the concept of texture samples assigned to pixel centers, where the pixel is considered to be a solid square rather than a single point location. My choice for the B-spline curve parameter is $t \in [-1/2, c+1/2]$. The mapping from t to s is $s = d + ((c+1-d)/(c+1))(t+1/2)$. The B-spline curve is $\mathbf{X}(t)$. Compute the B-spline curve point $\mathbf{X}(i)$ to approximate the control point $\mathbf{F}_i$ in the sense that given a sequence of input control points $\mathbf{F}_i$, you can produce a sequence of (smoothed) output control points $\mathbf{G}_i = \mathbf{X}(i)$.

4 Polynomial Evaluation using Horner's Method

Let $p(u) = \sum_{k=0}^{d} p_k u^k$ be a polynomial of degree d with $p_d \neq 0$. The naive evaluation algorithm treats the polynomial as a dot product of coefficients and powers of u,

$$p(u) = (p_0, p_1, \ldots, p_d) \cdot (1, u, \ldots, u^d) \tag{24}$$

This is not a good choice when using floating-point arithmetic; for example, if u is a small number and d is somewhat large, the powers might eventually become zero (before reaching power d) due to floating-point rounding errors. A more robust approach for evaluation of $p(u)$ is Horner's method,

$$p(u) = p_0 + u(p_1 + u(p_2 \ldots) \ldots) \tag{25}$$

For example, a degree-2 polynomial factors as $p(u) = p_0 + u(p_1 + up_2)$ and a degree-3 polynomial factors as $p(u) = p_0 + u(p_1 + u(p_2 + up_3))$. In the context of B-spline derivative evaluation, the polynomials are of the form $p(u) = (a_0 b_0, a_1 b_1, \ldots, a_d b_d) \cdot (1, u, \ldots, u^d)$, in which case the premultiplications $a_i b_i$ increase the computational cost slightly. Generally, Horner's method is shown in Listing 2. The listing also includes the variation mentioned previously.

Listing 2. Pseudocode for Horner's method to evaluate a polynomial robustly.

```
int const degree = some_positive_degree;
double p[degree + 1];  // the polynomial coefficients
double u;  // input to the polynomial

// one evaluation scheme, avoids some initialization code
double value = 0.0;
for (int i = degree; i >= 0; --i)
{
  value = p[i] + u * value;
}

// another evaluation scheme with additional initialization code
int i = degree;
double value = p[degree];
for (--i; i >= 0; --i)
{
  value = p[i] + u * value;
}

// The variation when the polynomial has coefficients a[i]*b[i].
int const degree = some_positive_degree;
double a[degree + 1], b[degree + 1];
double u;
double value = 0.0;
for (int i = degree, i >= 0, --i)
{
  value = a[i] * b[i] + u * value;
}
```

The hope is that each intermediate value is of appropriate size so that even when u is small, p[i] + u * value does not have significant rounding errors. The operation count for a single evaluation of a degree-d polynomial is $2d$ because there are d multiplications and d additions. In the variation where the coefficients are products of numbers, the operation count for a single evaluation is $3d + 1$ because there are $d + 1$ premultiplications followed by the $2d$ operations for the pairs of multiplications and additions in the for-loop.

5 B-Splines for 1-Dimensional Data

Let the 1-dimensional samples be $\{\mathbf{F}_i\}_{i=0}^{c}$ and choose the degree $d > 0$ for the B-spline interpolator. The B-spline interpolator is that of equation (22),

$$\mathbf{X}_{d+i}(s) = \sum_{j=0}^{d} \mathbf{F}_{i+j} \sum_{k=0}^{d} a_{jk}^{(d)} (s-d-i)^k \tag{26}$$

for $0 \le i \le c-d$ and where $s \in [d+i, d+i+1)$. Keep in mind that the implementation will require you to choose $t \in [-1/2, c+1/2]$ and then compute internally $s-d = ((c+1-d)/(c+1))(t+1/2)$. The index i is determined from s in order to select $\mathbf{X}_{d+i}(s)$ for evaluation.

Given a value t, the computation of i, s and $u = s-d-i$ from t is a constant-time operation. Listing 3 shows pseudocode for the operation.

Listing 3. Given $t \in [-1/2, c+1/2)$, determine the corresponding interval $[i, i+1)$ that contains $s-d$ and then compute $u = s-d-i$. The actual input t is clamped to the support interval $[-1/2, c+1/2]$, which includes the right endpoint $c+1/2$.

```
// Inputs:
//   tmin = -1/2, tmax = c + 1/2, dsdt = (c + 1 - d)/(c + 1), numControls = c + 1, degree = d
// Outputs:
//   i: the interval [i,i+1) contains s - d,  u = s - d - i in [0,1)
void GetKey(Real t, Real tmin, Real tmax, Real dsdt, int numControls, int degree, int& i, Real& u)
{
    // Compute s - d = ((c + 1 - d)/(c + 1))(t + 1/2) and the index i for which
    // d + i <= s <= d + i + 1.  Let u = s - d - i so that 0 <= u <= 1.
    if (t > tmin)
    {
        if (t < tmax)
        {
            Real s_minus_d = dsdt * (t - tmin);  // s - d
            i = floor(s_minus_d);  // largest integer smaller or equal to s - d
            u = s_minus_d - i;  // s - d - i in [0,1)
        }
        else  // clamp to t = tmax
        {
            // When t = tmax, s - d = c + 1 - d.  Choosing i = floor(s - d) = c + 1 - d as in
            // the case t < tmax leads to an interval [c + 1 - d, c + 2 - d) that is outside the
            // domain of s - d.  Choose instead i = c - d and u = 1, which corresponds to the
            // right endpoint of the support of the t-domain.
            i = numControls - 1 - degree;  // c - d
            u = 1;
        }
    }
    else  // clamp to t = tmin
    {
        i = 0;
        u = 0;
    }
}
```

The computational costs for the B-spline evaluation depend on the distribution of the s-values. Several evaluation methodologies are described next.

5.1 Evaluation without Caching

The straightforward evaluation algorithm is to compute the polynomial terms first and then compute the weighted sum of control points. Equation (26) is parenthesized as follows for this algorithm,

$$\mathbf{X}_{d+i}(s) = \sum_{j=0}^{d} \mathbf{F}_{i+j} \left(\sum_{k=0}^{d} a_{jk}^{(d)} u^k \right) = \sum_{j=0}^{d} \mathbf{F}_{i+j} \phi_j(u) \tag{27}$$

where $u = s - d - i \in [0, 1)$ and where $\phi_j(u)$ is the dot product of row j of $A^{(d)}$ with the polynomial vector $(1, u, \ldots, u^d)$. The polynomials should be evaluated using Horner's method as defined in equation (25).

The operation count for a single evaluation of a degree-d polynomial is $2d$ because there are d multiplications and d additions. There are $d+1$ polynomials $\phi_j(u)$ to evaluate, so the collective operation count is $2d(d+1)$. Although the motivation for this document is that the control points $\mathbf{F}_i$ are scalars, the interpolation applies for higher-dimensional data; let α be the dimension of the control points. The weighted average of the control points requires $d+1$ multiplications of scalars times control points and d additions of those products. The total operation count for an evaluation with a single u is $2d(d+1) + \alpha(2d+1)$.

The B-spline derivative of order m is

$$\mathbf{X}_{d+i}^{(m)}(s) = \sum_{j=0}^{d} \mathbf{F}_{i+j} \left(\sum_{k=0}^{d-m} a_{j,(k+m)}^{(d)} w_{mk} u^k \right) = \sum_{j=0}^{d} \mathbf{F}_{i+j} \phi_j^{(m)}(u) \tag{28}$$

where

$$w_{mk} = (k+1) \cdots (k+m) = \prod_{\ell=1}^{m} (k+\ell), \quad 1 \le m \le d, \; 0 \le k \le d-m \tag{29}$$

and where $\phi_j^{(m)}(u)$ is the derivative of order m of $\phi_j(u)$. Define $w_{0k} = 1$ for $0 \le k \le d$ so that $\phi_j(u) = \phi_j^{(0)}(u)$, effectively extending equation (29) for $0 \le m \le d$.

The w_{mk} can be precomputed and stored in a 1-dimensional array. For example, when $d = 3$, we can precompute

$$\left[\; w_{00} \;\; w_{01} \;\; w_{02} \;\; w_{03} \,\middle|\, w_{10} \;\; w_{11} \;\; w_{12} \,\middle|\, w_{20} \;\; w_{21} \,\middle|\, w_{30} \; \right] = \left[\; 1 \;\; 1 \;\; 1 \;\; 1 \,\middle|\, 1 \;\; 2 \;\; 3 \,\middle|\, 2 \;\; 6 \,\middle|\, 6 \; \right] \tag{30}$$

As it turns out, we only need to know the location of the last coefficient corresponding to order m. It is $L_m = (m+1)d - m(m-1)/2$ for $0 \le m \le d$. Starting with this location, Horner's method allows us to decrement the location index to discover each required coefficient. In the example when $d = 3$ these indices are $L_0 = 3$, $L_1 = 6$, $L_2 = 8$ and $L_3 = 9$. Listing 4 contains pseudocode for precomputing the w_{mk} and the location indices.

Listing 4. Pseudocode for precomputing the cofficients that store products of powers for the derivative evaluations.

```
void ComputeDCoefficients(int d, Real dCoefficients[(d+1)(d+2)/2], int lmax[d+1])
{
    // Compute w{0,k} corresponding to order m = 0.  This includes initialization of all w-terms
    // to 1 because for m > 0, we will incrementally multiply the w-terms to obtain the derivative
    // coefficients.
    for (int i = 0; i < (d+1)(d+2)/2; ++i)
```

```
    {
        dCoefficients[i] = 1;
    }

    // Compute w{m,k} corresponding to order m > 0.
    for (int m = 1, i0 = 0, i1 = d + 1; m <= d; ++m)
    {
        ++i0;
        for (int j = m, factor = 1; j <= d; ++j, ++factor, ++i0, ++i1)
        {
            dCoefficients[i1] = dCoefficients[i0] * factor;
        }
    }

    // Compute the locations of the last coefficients in each order's block.
    lmax[0] = d;
    for (int i0 = 0, i1 = 1; i1 <= d; i0 = i1++)
    {
        lmax[i1] = lmax[i0] + d - i0;
    }
}
```

A similar cost analysis applies to derivatives of order m of the B-spline function with $1 \le m \le d$. To compute the B-spline derivative of equation (28), Horner's method is used for evaluating the $d+1$ polynomials $\phi_j^{(m)}(u)$. A single polynomial evaluation requires $3(d-m)+1$ operations and the weighted sum of control points requires $\alpha(2d+1)$ operations, leading to a total operation count of $(3(d-m)+1)(d+1)+\alpha(2d+1)$. Pseudocode for evaluating equation (28) is shown in Listing 5.

Listing 5. Pseudocode for evaluating the 1-dimensional B-spline or of its derivatives as shown in equation (27). The Controls::Type is the type of the control points $\mathbf{F}_i$ and ctZero is the natural zero value for that type. The pseudocode for the function GetKey is found in Listing 3. The evaluation is in terms of the t-variable, but equation (28) is in terms of the s-variable where $s = d+((c+1-d)/(c+1))(t+1/2)$. Using the chain rule when differentiating, we need to multiply the specified-order derivative by a power of $ds/dt = (c+1-d)/(c+1)$.

```
//  degree: degree of the B-spline (d)
//  numControls: number of control points (c+1)
//  controls: array of control points
//  tmin:  -1/2
//  tmax:  c + 1/2
//  powerDSDT:  array of powers of ds/dt = (c + 1 - d) / (c + 1)
//  dcoefficient:  array of derivative polynomial coefficients, size (d+1)*(d+2)/2
//  lmax:  location of the last coefficient for the block of specified order
//  A: blending matrix, accessed as A(row,col)
//  phi:  array of polynomial evaluations that multiply the control points

Controls::Type EvaluateNoCaching(int order, Real t)
{
    Controls::Type result = ctZero;
    if (0 <= order && order <= degree)
    {
        int i;
        Real u;
        GetKey(t, tmin, tmax, powerDSDT[1], numControls, degree, i, u);

        for (int j = 0; j <= degree; ++j)
        {
            phi[j] = 0;
            for (int k = degree, ell = lmax[order]; k >= order; --k, --ell)
            {
                phi[j] = phi[j] * u + A(j, k) * dcoefficient[ell];
            }
        }
```

```
        for (int j = 0; j <= degree; ++j)
        {
            result = result + controls(i + j) * phi[j];
        }

        result = result * powerDSDT[order];
    }
    return result;
}
```

5.2 Evaluation with Precaching

In some applications there will be many evaluations involving s-values that lead to the same index i. The total cost of the evaluations can be reduced from that of the previous section by precomputing a matrix of blended control points. Equation 26 is rewritten as follows for this algorithm,

$$\mathbf{X}_{d+i}(s) = \sum_{j=0}^{d} \mathbf{F}_{i+j} \sum_{k=0}^{d} a_{jk}^{(d)} u^k = \sum_{k=0}^{d} \left(\sum_{j=0}^{d} \mathbf{F}_{i+j} a_{jk}^{(d)} \right) u^k = \sum_{k=0}^{d} \mathbf{\Phi}_{ik} u^k \tag{31}$$

where the last equality defines the $(c+1-d) \times (d+1)$ vector-valued matrix $\mathbf{\Phi}$ with elements $\mathbf{\Phi}_{ik}$ for $0 \le i \le c-d$ and $0 \le k \le d$. Observe that the order of the j- and k-summations is reversed.

For a selected i and corresponding u, $\sum_{k=0}^{d} \mathbf{\Phi}_{ik} u^k$ can be computed using Horner's method. This requires $2d$ operations, d multiplications of scalars and blended control points and d additions of those products, for a total of $2\alpha d$ scalar operations where α is the dimension of the control points. The operation count using the no-cached algorithm is $2d(d+1)+\alpha(2d+1)$, so the precached algorithm is less expensive to compute at runtime.

Precomputing of $\mathbf{\Phi}$ is assumed to be offline, so its cost is not included in the operation count. Each matrix element involves $d+1$ products of scalars times control points followed by d additions of those products for a cost of $\alpha(2d+1)$ operations. The total cost of the precomputing is $\alpha(2d+1)(d+1)(c+1-d)$. The dominant cost occurs because of the potentially large number of control points. The precaching algorithm has an expensive—but fixed—offline cost that is justified when the number of runtime evaluations is sufficiently large. Precaching also has a space–time tradeoff. The number of blended control points is $(d+1)(c+1-d)$, which is larger than the number of control points $c+1$. The algorithm uses additional memory in order to obtain better runtime performance. Listing 6 contains pseudocode for precomputing the elements of $\mathbf{\Phi}$.

Listing 6. Pseudocode for precomputing $\mathbf{\Phi}_{ik} = \sum_{j=0}^{d} \mathbf{F}_{i+j} a_{jk}^{(d)}$ for all i and k with $0 \le i \le c-d$ and $0 \le k \le d$.

```
//  degree: degree of the B-spline (d)
//  numControls: number of control points (c+1)
//  controls: array of control points
//  A: blending matrix, accessed as A(row,col)
//  tensor: blended controls, accessed as tensor(row,col)

void ComputeTensor(int r, int c)
{
    Controls::Type element = ctZero;
    for (int j = 0; j <= degree; ++j)
    {
        element = element + controls(r + j) * A(j, c);
```

```
    }
    tensor(r, c) = element;
}

void InitializeTensors()
{
    int numRows = numControls - degree;
    int numCols = degree + 1;
    for (int r = 0; r < numRows; ++r)
    {
        for (int c = 0; c < numCols; ++c)
        {
            ComputeTensor(r, c);
        }
    }
}
```

The B-spline derivative of order m when using the precaching algorithm is

$$\mathbf{X}_{d+i}^{(m)}(s) = \sum_{k=0}^{d-m} \left(\mathbf{\Phi}_{i,k+m} w_{mk}\right) u^k \tag{32}$$

where w_{mk} is defined in equation (29). The evaluation uses Horner's method where the coefficients are products of tensor elements with scalars. Listing 7 contains pseudocode for the evaluation of equation (32) using the summation involving $\mathbf{\Phi}_{i,k+m}$.

Listing 7. Pseudocode for the evaluation of equation (32).

```
//  degree: degree of the B-spline (d)
//  numControls: number of control points (c+1)
//  controls: array of control points
//  tmin:  -1/2
//  tmax:  c + 1/2
//  powerDSDT:  array of powers of ds/dt = (c + 1 - d) / (c + 1)
//  dcoefficient:  array of derivative polynomial coefficients, size (d+1)*(d+2)/2
//  lmax:  location of the last coefficient for the block of specified order
//  tensor: blended controls, accessed as tensor(row,col)

Controls::Type EvaluatePrecaching(int order, Real t)
{
    Controls::Type result = ctZero;
    if (0 <= order && order <= degree)
    {
        int i;
        Real u;
        GetKey(t, tmin, tmax, powerDSDT[1], numControls, degree, i, u);

        for (int k = degree, ell = lmax[order]; k >= order; --k, --ell)
        {
            result = result * u + tensor(i, k) * dcoefficient[ell];
        }

        result = result * powerDSDT[order];
    }
    return result;
}
```

5.3 Evaluation with On-Demand Caching

In this methodology, the elements of the matrix $\mathbf{\Phi}$ are not precomputed; rather, they are computed only when needed during an evaluation. Once they are computed, they are stored in the tensor(r,c) cache. A

matrix of Boolean flags, cached(r,c), is maintained indicating whether or not a matrix element needs to be computed. The flags are all set to false initially. The cost of precomputing all the matrix elements is not incurred. Instead, the cost is based only on computing those elements that are needed during the application execution. This is useful when the application focuses only on a region of interest involving a subset of the control points. Listing 8 contains pseudocode for the evaluation.

Listing 8. Pseudocode for the evaluation of equation (32). The **Φ** elements are computed the first time they are needed and the results are stored in a cache.

```
//  degree: degree of the B-spline (d)
//  numControls: number of control points (c+1)
//  controls: array of control points
//  tmin:  -1/2
//  tmax:  c + 1/2
//  powerDSDT:  array of powers of ds/dt = (c + 1 - d) / (c + 1)
//  dcoefficient:  array of derivative polynomial coefficients, size (d+1)*(d+2)/2
//  lmax:  location of the last coefficient for the block of specified order
//  tensor: blended controls, accessed as tensor(row,col)
//  cached:  flags for blending, accessed as cached(row,col)

Controls::Type EvaluateOnDemandCaching(int order, Real t)
{
    Controls::Type result = ctZero;
    if (0 <= order && order <= degree)
    {
        int i;
        Real u;
        GetKey(t, tmin, tmax, powerDSDT[1], numControls, degree, i, u);

        for (int k = degree, ell = lmax[order]; k >= order; --k, --ell)
        {
            if (cached(i, k) == false)
            {
                ComputeTensor(i, k);
                cached(i, k) = true;
            }
            result = result * u + tensor(i, k) * dcoefficient[ell];
        }

        result = result * powerDSDT[order];
    }
    return result;
}
```

6 B-Splines for 2-Dimensional Data

Let the 2-dimensional samples be $\{\mathbf{F}_{i_0,i_1}\}_{i_0=0,i_1=0}^{c_0,c_1}$ and choose the degrees $d_0 > 0$ and $d_1 > 0$ for the B-spline interpolator. It is not necessary that $d_0 = d_1$, although in some applications it is common to have the same degree. The B-spline interpolator is defined as a tensor product spline,

$$\mathbf{X}_{d_0+i_0,d_1+i_1}(s_0,s_1) = \sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1} F_{i_0+j_0,i_1+j_1} \sum_{k_0=0}^{d_0}\sum_{k_1=0}^{d_1} a_{j_0k_0}^{(d_0)} a_{j_1k_1}^{(d_1)} (s_0-d_0-i_0)^{k_0}(s_1-d_1-i_1)^{k_1} \tag{33}$$

for $0 \le i_n \le c_n - d_n$ and $s_n \in [d_n + i_n, d_n + i_n + 1)$. The implementation will require you to choose $t_n \in [-1/2, c_n + 1/2]$ and then compute internally $s_n - d_n = ((c_n + 1 - d_n)/(c_n + 1))(t_n + 1/2)$. The indices i_n are determined from s_n in order to select $\mathbf{X}_{d_0+i_0,d_1+i_1}(s_0,s_1)$ for evaluation. The evaluation methodologies parallel those for the 1-dimensional B-spline functions.

6.1 Evaluation without Caching

The straightforward evaluation algorithm is to compute the polynomial terms first and then compute the weighted sum of control points. Equation (33) is parenthesized as follows for this algorithm,

$$\begin{aligned}\mathbf{X}_{d_0+i_0,d_1+i_1}(s_0,s_1) &= \sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1} F_{i_0+j_0,i_1+j_1}\left(\sum_{k_0=0}^{d_0} a_{j_0k_0}^{(d_0)}u_0^{k_0}\right)\left(\sum_{k_1=0}^{d_1} a_{j_1k_1}^{(d_1)}u_1^{k_1}\right)\\ &= \sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1} F_{i_0+j_0,i_1+j_1}\phi_{0,j_0}(u_0)\phi_{1,j_1}(u_1)\end{aligned} \tag{34}$$

where $u_n = s_n - d_n - i_n \in [0,1)$ and $\phi_{n,j_n}(u_n)$ is the dot product of row j_n of $A^{(d_n)}$ with the polynomial vector $(1, u_n, \ldots, u_n^{d_n})$.

The B-spline derivatives of order (m_0, m_1) are

$$\mathbf{X}_{d_0+i_0,d_1+i_1}^{(m_0,m_1)}(s_0,s_1) = \sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1} F_{i_0+j_0,i_1+j_1}\phi_{0,j_0}^{(m_0)}(u_0)\phi_{1,j_1}^{(m_1)}(u_1) \tag{35}$$

where

$$\phi_{n,j_n}^{(m_n)}(u_n) = \sum_{k_n=0}^{d_n-m_n} a_{j_n,(k_n+m_n)}^{(d_n)} w_{m_n,k_n}^{(d_n)} u_n^{k_n} \tag{36}$$

are the derivatives of $\phi_{n,j_n}(u_n)$ with $w_{m_n,k_n}^{(d_n)}$ provided by equation (29). The superscript on the w-term emphasizes that the subscripts of that term have constraints that depend on the degree d_n.

The evaluation of equation (35) is a simple extension of that shown in equation (28). Pseudocode is provided in Listing 9.

Listing 9. Evaluation of the 2-dimensional B-spline or of its derivatives as shown in equation (35).

```
//  degree[2]: degrees of the B-spline
//  numControls[2]: numbers of control point
//  controls: array of control points
//  tmin[2]:  -1/2
//  tmax[2]:  (numControls[] + 1/2 )
//  powerDSDT[2]:  array of powers of ds/dt = (numControls[] - d) / numControls[]
//  dcoefficient[2]:  array of derivative polynomial coefficients, size (d[]+1)*(d[]+2)/2
//  lmax[2]:  location of the last coefficient for the block of specified order
//  A[2]: blending matrices, accessed as A[](row,col)
//  phi[2]:  array of polynomial evaluations that multiply the control points

Controls::Type EvaluateNoCaching(int order[2], Real t[2])
{
    Controls::Type result = ctZero;
    if (0 <= order[0] && order[0] <= degree[0] &&
        0 <= order[1] && order[1] <= degree[1])
    {
        int i[2];
        Real u[2];
        for (int dim = 0; dim < 2; ++dim)
        {
            GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
                i[dim], u[dim]);
        }

        for (int dim = 0; dim < 2; ++dim)
        {
            for (int j = 0; j <= degree[dim]; ++j)
```

```
            {
                phi[dim][j] = 0;
                for (int k = degree[dim], ell = lmax[dim][order[dim]]; k >= order[dim]; --k, --ell)
                {
                    phi[dim][j] = phi[dim][j] * u[dim] + A[dim](j, k) * dcoefficient[dim][ell];
                }
            }
        }

        for (int j1 = 0; j1 <= degree[1]; ++j1)
        {
            Real phi1 = phi[1][j1];
            for (int j0 = 0; j0 <= degree[0]; ++j0)
            {
                Real phi0 = phi[0][j0];
                Real phi01 = phi0 * phi1;
                result = result + controls(i[0] + j0, i[1] + j1) * phi01;
            }
        }

        Real adjust = 1;
        for (int dim = 0; dim < 2; ++dim)
        {
            adjust *= powerDSDT[dim][order[dim]];
        }
        result = result * adjust;
    }
    return result;
}
```

6.2 Evaluation with Precaching

Equation (33) is rewritten as follows for this algorithm,

$$\begin{aligned} \mathbf{X}_{d_0+i_0,d_1+i_1}(s_0,s_1) &= \sum_{k_0=0}^{d_0}\sum_{k_1=0}^{d_1}\left(\sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1} F_{i_0+j_0,i_1+j_1} a_{j_0k_0}^{(d_0)} a_{j_1k_1}^{(d_1)}\right) u_0^{k_0} u_1^{k_1} \\ &= \sum_{k_1=0}^{d_1}\left(\sum_{k_0=0}^{d_0} \mathbf{\Phi}_{i_0i_1k_0k_1} u_0^{k_0}\right) u_1^{k_1} \end{aligned} \tag{37}$$

and its derivatives of order (m_0, m_1) are

$$\mathbf{X}_{d_0+i_0,d_1+i_1}^{(m_0,m_1)}(s_0,s_1) = \sum_{k_1=0}^{d_1-m_1}\left(\sum_{k_0=0}^{d_0-m_0} \mathbf{\Phi}_{i_0,i_1,k_0+m_0,k_1+m_1} w_{m_0,k_0}^{(d_n)} u_0^{k_0}\right) w_{m_1,k_1}^{(d_1)} u_1^{k_1} \tag{38}$$

Listing 10 contains pseudocode for precomputing the tensor $\mathbf{\Phi}_{i_0i_1k_0k_1}$.

Listing 10. Pseudocode for precomputing $\mathbf{\Phi}_{r_0r_1c_0c_1}$. The index names r and c are suggestive of the ordering for the tensor in the 1-dimensional B-spline precache algorithm.

```
//  degree[2]: degrees of the B-spline
//  numControls[2]: numbers of control point
//  controls: array of control points
//  A[2]: blending matrices, accessed as A[](row,col)
//  tensor: blended controls, accessed as tensor(r0,r1,c0,c1)

void ComputeTensor(int r0, int r1, int c0, int c1)
{
    Controls::Type element = ctZero;
    for (int j1 = 0; j1 <= degree[1]; ++j1)
    {
        Real blend1 = A[1](j1, c1);
```

```
        for (int j0 = 0; j0 <= degree[0]; ++j0)
        {
            Real blend0 = A[0](j0, c0);
            Real blend01 = blend0 * blend1;
            element = element + controls(r0 + j0, r1 + j1) * blend01;
        }
    }
    tensor(r0, r1, c0, c1) = element;
}

void InitializeTensors()
{
    int numRows[2], numCols[2];
    for (int dim = 0; dim < 2; ++dim)
    {
        numRows[dim] = numControls[dim] - degree[dim];
        numCols[dim] = degree[dim] + 1;
    }
    for (int r1 = 0; r1 < numRows[1]; ++r1)
    {
        for (int r0 = 0; r0 <numRows[0]; ++r0)
        {
            for (int c1 = 0; c1 < numCols[1]; ++c1)
            {
                for (int c0 = 0; c0 < numCols[0]; ++c0)
                {
                    ComputeTensor(r0, r1, c0, c1);
                }
            }
        }
    }
}
```

Listing 11 contains pseudocode for the evaluation of equation (37).

Listing 11. Pseudocode for evaluation the 2-dimensional B-spline function using precaching.

```
//  degree[2]: degrees of the B-spline
//  numControls[2]: numbers of control point
//  controls: array of control points
//  tmin[2]:  -1/2
//  tmax[2]:  (numControls[] + 1/2 )
//  powerDSDT[2]:  array of powers of ds/dt = (numControls[] - d) / numControls[]
//  dcoefficient[2]:  array of derivative polynomial coefficients, size (d[]+1)*(d[]+2)/2
//  lmax[2]:  location of the last coefficient for the block of specified order
//  tensor: blended controls, accessed as tensor(r0,r1,c0,c1)

Controls::Type EvaluatePrecaching(int order[2], Real t[2])
{
    Controls::Type result = ctZero;
    if (0 <= order[0] && order[0] <= degree[0] &&
        0 <= order[1] && order[1] <= degree[1])
    {
        int i[2];
        Real u[2];
        for (int dim = 0; dim < 2; ++dim)
        {
            GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
                i[dim], u[dim]);
        }

        for (int k1 = degree[1], ell1 = lmax[1][order[1]]; k1 >= order[1]; --k1, --ell1)
        {
            Controls::Type term = ctZero;
            for (int k0 = degree[0], ell0 = lmax[0][order[0]]; k0 >= order[0]; --k0, --ell0)
            {
                term = term * u[0] + tensor(i0, i1, k0, k1) * dcoefficient[0][ell0];
```

```
            }
            result = result * u[1] + term * dcoefficient[1][ell1];
        }

        Real adjust(1);
        for (int dim = 0; dim < 2; ++dim)
        {
            adjust *= powerDSDT[dim][order[dim]];
        }
        result = result * adjust;
    }
    return result;
}
```

6.3 Evaluation with On-Demand Caching

As in the 1-dimensional case, the tensor elements $\mathbf{\Phi}_{r_0 r_1 c_0 c_1}$ are computed for the first time they are needed and then cached for later access if necessary. A tensor of Boolean flags, cached(r0,r1,c0,c1), is maintained indicating whether or not a tensor element needs to be computed. The flags are all set to false initially. The cost of precomputing all the matrix elements is not incurred. Instead, the cost is based only on computing those elements that are needed during the application execution. This is useful when the application focuses only on a region of interest involving a subset of the control points. Listing 12 contains pseudocode for the evaluation.

Listing 12. Pseudocode for the evaluation of equation (38). The $\mathbf{\Phi}$ elements are computed the first time they are needed and the results are stored in a cache.

```
//  degree[2]: degrees of the B-spline
//  numControls[2]: numbers of control point
//  controls: array of control points
//  tmin[2]:  -1/2
//  tmax[2]:  (numControls[] + 1/2 )
//  powerDSDT[2]:  array of powers of ds/dt = (numControls[] - d) / numControls[]
//  dcoefficient[2]:  array of derivative polynomial coefficients, size (d[]+1)*(d[]+2)/2
//  lmax[2]:  location of the last coefficient for the block of specified order
//  tensor: blended controls, accessed as tensor(r0,r1,c0,c1)
//  cached:  flags for blending, accessed as cached(r0,r1,c0,c1)

Controls::Type EvaluateOnDemandCaching(int order[2], Real t[2])
{
    Controls::Type result = ctZero;
    if (0 <= order[0] && order[0] <= degree[0] &&
        0 <= order[1] && order[1] <= degree[1])
    {
        int i[2];
        Real u[2];
        for (int dim = 0; dim < 2; ++dim)
        {
            GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
                i[dim], u[dim]);
        }

        for (int k1 = degree[1], ell1 = lmax[1][order[1]]; k1 >= order[1]; --k1, --ell1)
        {
            Controls::Type term = ctZero;
            for (int k0 = degree[0], ell0 = lmax[0][order[0]]; k0 >= order[0]; --k0, --ell0)
            {
                if (cached(i0, i1, k0, k1) == false)
                {
                    ComputeTensor(i0, i1, k0, i1);
                    cached(i0, i1, k0, k1) = true;
```

```
                }
                term = term * u[0] + tensor(i0, i1, k0, k1) * dcoefficient[0][ell0];
            }
            result = result * u[1] + term * dcoefficient[1][ell1];
        }

        Real adjust(1);
        for (int dim = 0; dim < 2; ++dim)
        {
            adjust *= powerDSDT[dim][order[dim]];
        }
        result = result * adjust;
    }
    return result;
}
```

7 B-Splines for 3-Dimensional Data

Let the 3-dimensional samples be $\{\mathbf{F}_{i_0,i_1,i_2}\}_{i_0=0,i_1=0,i_2}^{c_0,c_1,c_2}$ and choose the degrees $d_0 > 0$, $d_1 > 0$ and $d_2 > 0$ for the B-spline interpolator. It is not necessary that the degrees all be the same value, although in some applications it is common to have the same degree. The B-spline interpolator is defined as a tensor product spline,

$$\begin{aligned}\mathbf{X}_{d_0+i_0,d_1+i_1,d_2+i_2}(s_0,s_1,s_2) &= \textstyle\sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1}\sum_{j_2=0}^{d_1} F_{i_0+j_0,i_1+j_1,i_2+j_2} \\ &\quad \textstyle\sum_{k_0=0}^{d_0}\sum_{k_1=0}^{d_1}\sum_{k_2=0}^{d_2} a_{j_0k_0}^{(d_0)} a_{j_1k_1}^{(d_1)} a_{j_2k_2}^{(d_2)} (s_0-d_0-i_0)^{k_0}(s_1-d_1-i_1)^{k_1}(s_2-d_2-i_2)^{k_2}\end{aligned} \tag{39}$$

for $0 \le i_n \le c_n - d_n$ and $s_n \in [d_n + i_n, d_n + i_n + 1)$. The implementation will require you to choose $t_n \in [-1/2, c_n + 1/2]$ and then compute internally $s_n - d_n = ((c_0 + 1 - d_0)/(c_0 + 1))(t_0 + 1/2)$. The indices i_n are determined from s_n in order to select $\mathbf{X}_{d_0+i_0,d_1+i_1,d_2+i_2}(s_0, s_1, s_2)$ for evaluation. The evaluation methodologies parallel those for the 1-dimensional B-spline functions.

7.1 Evaluation without Caching

The straightforward evaluation algorithm is to compute the polynomial terms first and then compute the weighted sum of control points. Equation (39) is parenthesized as follows for this algorithm,

$$\begin{aligned}&\mathbf{X}_{d_0+i_0,d_1+i_1,d_2+i_2}(s_0,s_1,s_2) \\ &\quad = \textstyle\sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1}\sum_{j_2=0}^{d_2} F_{i_0+j_0,i_1+j_1,i_2+j_2} \left(\sum_{k_0=0}^{d_0} a_{j_0k_0}^{(d_0)} u_0^{k_0}\right)\left(\sum_{k_1=0}^{d_1} a_{j_1k_1}^{(d_1)} u_1^{k_1}\right)\left(\sum_{k_2=0}^{d_2} a_{j_2k_2}^{(d_2)} u_2^{k_2}\right) \\ &\quad = \textstyle\sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1}\sum_{j_2=0}^{d_2} F_{i_0+j_0,i_1+j_1,i_2+j_2}\phi_{0,j_0}(u_0)\phi_{1,j_1}(u_1)\phi_{2,j_2}(u_2)\end{aligned} \tag{40}$$

where $u_n = s_n - d_n - i_n \in [0, 1)$ and $\phi_{n,j_n}(u_n)$ is the dot product of row j_n of $A^{(d_n)}$ with the polynomial vector $(1, u_n, \ldots, u_n^{d_n})$.

The B-spline derivatives of order (m_0, m_1, m_2) are

$$\mathbf{X}_{d_0+i_0,d_1+i_1,d_2+i_2}^{(m_0,m_1,m_2)}(s_0,s_1,s_2) = \textstyle\sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1}\sum_{j_2=0}^{d_2} F_{i_0+j_0,i_1+j_1,i_2+j_2}\phi_{0,j_0}^{(m_0)}(u_0)\phi_{1,j_1}^{(m_1)}(u_1)\phi_{2,j_2}^{(m_2)}(u_2) \tag{41}$$

where

$$\phi_{n,j_n}^{(m_n)}(u_n) = \sum_{k_n=0}^{d_n-m_n} a_{j_n,(k_n+m_n)}^{(d_n)} w_{n,m_nk_n}^{(d_n)} u_n^{k_n} \tag{42}$$

are the polynomial derivatives of $\phi_{n,j_n}(u_n)$ with $w^{(d_n)}_{m_n,k_n}$ provided by equation (29). The superscript on the w-term emphasizes that the subscripts of that term have constraints that depend on the degree d_n.

The evaluation of equation (41) is a simple extension of that shown in equation (28). Pseudocode is provided in Listing 13.

Listing 13. Evaluation of the 3-dimensional B-spline or of its derivatives as shown in equation (41).

```
//  degree[3]: degrees of the B-spline
//  numControls[3]: numbers of control point
//  controls: array of control points
//  A[3]: blending matrix of size (d[]+1)-by-(d[]+1), stored as A[](row,col)
//  dcoefficient[3]:  array of derivative polynomial coefficients, size (d[]+1)*(d[]+2)/2
//  lmax[3]:  location of the last coefficient for the block of specified order
//  phi[3]:  array of polynomial evaluations that multiply the control points
//  tmin[3]:  -1/2
//  tmax[3]:  (numControlPoints[] + 1/2 )
//  powerDSDT[3]:  array of powers of ds/dt = (numControls[] - d) / numControls[]

Controls::Type EvaluateNoCaching(int order[3], Real t[3])
{
    Controls::Type result = ctZero;
    if (0 <= order[0] && order[0] <= degree[0] &&
        0 <= order[1] && order[1] <= degree[1] &&
        0 <= order[2] && order[2] <= degree[2])
    {
        int i[3];
        Real u[3];
        for (int dim = 0; dim < 3; ++dim)
        {
            GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
                i[dim], u[dim]);
        }

        for (int dim = 0; dim < 3; ++dim)
        {
            for (int j = 0; j <= degree[dim]; ++j)
            {
                phi[dim][j] = 0;
                for (int k = degree[dim], ell = lmax[dim][order[dim]]; k >= order[dim]; --k, --ell)
                {
                    phi[dim][j] = phi[dim][j] * u[dim] + A[dim](j, k) * dcoefficient[dim][ell];
                }
            }
        }

        for (int j2 = 0; j2 <= degree[2]; ++j2)
        {
            Real phi2 = phi[2][j2];
            for (int j1 = 0; j1 <= degree[1]; ++j1)
            {
                Real phi1 = phi[1][j1];
                Real phi12 = phi1 * phi2;
                for (int j0 = 0; j0 <= degree[0]; ++j0)
                {
                    Real phi0 = phi[0][j0];
                    Real phi012 = phi0 * phi12;
                    result = result + controls(i[0] + j0, i[1] + j1, i[2] + j2) * phi012;
                }
            }
        }

        Real adjust = 1;
        for (int dim = 0; dim < 3; ++dim)
        {
            adjust *= powerDSDT[dim][order[dim]];
        }
        result = result * adjust;
```

```
    }
    return result;
}
```

7.2 Evaluation with Precaching

Equation (39) is rewritten as follows for this algorithm,

$$\begin{aligned}
&\mathbf{X}_{d_0+i_0,d_1+i_1,d_2+i_2}(s_0,s_1,s_2)\\
&\quad=\sum_{k_0=0}^{d_0}\sum_{k_1=0}^{d_1}\sum_{k_2=0}^{d_2}\left(\sum_{j_0=0}^{d_0}\sum_{j_1=0}^{d_1}\sum_{j_2=0}^{d_2}F_{i_0+j_0,i_1+j_1,i_2+j_2}a_{j_0k_0}^{(d_0)}a_{j_1k_1}^{(d_1)}a_{j_2k_2}^{(d_2)}\right)u_0^{k_0}u_1^{k_1}u_2^{k_2}\\
&\quad=\sum_{k_2=0}^{d_2}\left(\sum_{k_1=0}^{d_1}\left(\sum_{k_0=0}^{d_0}\mathbf{\Phi}_{i_0i_1i_2k_0k_1k_2}u_0^{k_0}\right)u_1^{k_1}\right)u_2^{k_2}
\end{aligned}\tag{43}$$

and its derivatives of order (m_0,m_1,m_2) are

$$\begin{aligned}
&\mathbf{X}^{(m_0,m_1,m_2)}_{d_0+i_0,d_1+i_1,d_2+i_2}(s_0,s_1,s_2)\\
&\quad=\sum_{k_2=0}^{d_2-m_2}\left(\sum_{k_1=0}^{d_1-m_1}\left(\sum_{k_0=0}^{d_0-m_0}\mathbf{\Phi}_{i_0,i_1,i_2,k_0+m_0,k_1+m_1,k_2+m_2}w_{m_0,k_0}^{(d_0)}u_0^{k_0}\right)w_{m_1,k_1}^{(d_1)}u_1^{k_1}\right)w_{m_2,k_2}^{(d_2)}u_2^{k_2}
\end{aligned}\tag{44}$$

Listing 14 contains pseudocode for precomputing the tensor $\mathbf{\Phi}_{i_0i_1i_2k_0k_1k_2}$.

Listing 14. Pseudocode for precomputing $\mathbf{\Phi}_{r_0r_1r_1c_0c_1c_2}$. The index names r and c are suggestive of the ordering for the tensor in the 1-dimensional B-spline precache algorithm.

```
//  degree[3]: degrees of the B-spline
//  numControls[3]: numbers of control point
//  controls: array of control points
//  A[3]: blending matrices, accessed as A[](row,col)
//  tensor: blended controls, accessed as tensor(r0,r1,r2,c0,c1,c2)

void ComputeTensor(int r0, int r1, int r2, int c0, int c1, int c2)
{
    Controls::Type element = ctZero;
    for (int j2 = 0; j2 <= degree[2]; ++j2)
    {
        Real blend2 = A[2](j2, c2);
        for (int j1 = 0; j1 <= degree[1]; ++j1)
        {
            Real blend1 = A[1](j1, c1);
            Real blend12 = blend1 * blend2;
            for (int j0 = 0; j0 <= degree[0]; ++j0)
            {
                Real blend0 = A[0](j0, c0);
                Real blend012 = blend0 * blend12;
                element = element + controls(r0 + j0, r1 + j1, r2 + j2) * blend012;
            }
        }
    }
    tensor(r0, r1, r2, c0, c1, c2) = element;
}

void InitializeTensors()
{
    int numRows[3], numCols[3];
    for (int dim = 0; dim < 3; ++dim)
    {
        numRows[dim] = numControls[dim] - degree[dim];
        numCols[dim] = degree[dim] + 1;
```

```
    }
    for (int r2 = 0; r2 < numRows[2]; ++r2)
    {
        for (int r1 = 0; r1 < numRows[1]; ++r1)
        {
            for (int r0 = 0; r0 <numRows[0]; ++r0)
            {
                for (int c2 = 0; c2 < numCols[2]; ++c2)
                {
                    for (int c1 = 0; c1 < numCols[1]; ++c1)
                    {
                        for (int c0 = 0; c0 < numCols[0]; ++c0)
                        {
                            ComputeTensor(r0, r1, r2, c0, c1, c2);
                        }
                    }
                }
            }
        }
    }
}
```

Listing 15 contains pseudocode for the evaluation of equation (43).

Listing 15. Pseudocode for evaluation the 3-dimensional B-spline function using precaching.

```
Controls::Type EvaluatePrecaching(int order[3], Real t[3])
{
    Controls::Type result = ctZero;
    if (0 <= order[0] && order[0] <= degree[0] &&
        0 <= order[1] && order[1] <= degree[1] &&
        0 <= order[2] && order[2] <= degree[2])
    {
        int i[3];
        Real u[3];
        for (int dim = 0; dim < 3; ++dim)
        {
            GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
                i[dim], u[dim]);
        }

        for (int k2 = degree[2], ell2 = lmax[2][order[2]]; k2 >= order[2]; --k2, --ell2)
        {
            Controls::Type term1 = ctZero;
            for (int k1 = degree[1], ell1 = lmax[1][order[1]]; k1 >= order[1]; --k1, --ell1)
            {
                Controls::Type term0 = ctZero;
                for (int k0 = degree[0], ell0 = lmax[0][order[0]]; k0 >= order[0]; --k0, --ell0)
                {
                    term0 = term0 * u[0] + tensor(i0, i1, i2, k0, k1, k2) * dcoefficient[0][ell0];
                }
                term1 = term1 * u[1] + term0 * dcoefficient[1][ell1];
            }
            result = result * u[2] + term1 * dcoefficient[2][ell2];
        }

        Real adjust = 1;
        for (int dim = 0; dim < 3; ++dim)
        {
            adjust *= powerDSDT[dim][order[dim]];
        }
        result = result * adjust;
    }
    return result;
}
```

7.3 Evaluation with On-Demand Caching

As in the 1-dimensional case, the tensor elements $\mathbf{\Phi}_{r_0 r_1 r_2 c_0 c_1 c_2}$ are computed for the first time they are needed and then cached for later access if necessary. A tensor of Boolean flags, cached(r0,r1,r2,c0,c1,c2), is maintained indicating whether or not a tensor element needs to be computed. The flags are all set to false initially. The cost of precomputing all the matrix elements is not incurred. Instead, the cost is based only on computing those elements that are needed during the application execution. This is useful when the application focuses only on a region of interest involving a subset of the control points. Listing 16 contains pseudocode for the evaluation.

Listing 16. Pseudocode for the evaluation of equation (38). The $\mathbf{\Phi}$ elements are computed the first time they are needed and the results are stored in a cache.

```
//  degree[3]: degrees of the B-spline
//  numControls[3]: numbers of control point
//  controls: array of control points
//  tmin[3]:  -1/2
//  tmax[3]:  (numControls[] + 1/2 )
//  powerDSDT[3]:  array of powers of ds/dt = (numControls[] - d) / numControls[]
//  dcoefficient[3]:  array of derivative polynomial coefficients, size (d[]+1)*(d[]+2)/2
//  lmax[3]:  location of the last coefficient for the block of specified order
//  tensor: blended controls, accessed as tensor(r0,r1,r2,c0,c1,c2)
//  cached:  flags for blending, accessed as cached(r0,r1,r2,c0,c1,c2)

Controls::Type EvaluateOnDemandCaching(int order[3], Real t[3])
{
    Controls::Type result = ctZero;
    if (0 <= order[0] && order[0] <= degree[0] &&
        0 <= order[1] && order[1] <= degree[1] &&
        0 <= order[2] && order[2] <= degree[2])
    {
        int i[3];
        Real u[3];
        for (int dim = 0; dim < 3; ++dim)
        {
            GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
                i[dim], u[dim]);
        }

        for (int k2 = degree[2], ell2 = lmax[2][order[2]]; k2 >= order[2]; --k2, --ell2)
        {
            Controls::Type term1 = ctZero;
            for (int k1 = degree[1], ell1 = lmax[1][order[1]]; k1 >= order[1]; --k1, --ell1)
            {
                Controls::Type term0 = ctZero;
                for (int k0 = degree[0], ell0 = lmax[0][order[0]]; k0 >= order[0]; --k0, --ell0)
                {
                    if (cached(i0, i1, i2, k0, k1, k2) == false)
                    {
                        ComputeTensor(i0, i1, i2, k0, k1, k2);
                        cached(i0, i1, i2, k0, k1, k2) = true;
                    }
                    term0 = term0 * u[0] + tensor(i0, i1, i2, k0, k1, k2) * dcoefficient[0][ell0];
                }
                term1 = term1 * u[1] + term0 * dcoefficient[1][ell1];
            }
            result = result * u[2] + term1 * dcoefficient[2][ell2];
        }

        Real adjust(1);
        for (int dim = 0; dim < 3; ++dim)
        {
            adjust *= powerDSDT[dim][order[dim]];
        }
        result = result * adjust;
```

```
    }
    return result;
}
```

8 B-Splines for Data in General Dimensions

The description here uses multiindex notation for dimension n. The multiindices are $\mathbf{i} = (i_0, \ldots, i_{n-1})$, $\mathbf{j} = (j_0, \ldots, j_{n-1})$, $\mathbf{c} = (c_0, \ldots, c_{n-1})$, $\mathbf{d} = (d_0, \ldots, d_{n-1})$ and $\mathbf{k} = (k_0, \ldots, k_{n-1})$. The multiindex for a tuple of all zeros is $\mathbf{0}$, the multiindex for a tuple of all ones is $\mathbf{1}$ and the multiindex for a tuple of all $1/2$ values is $\mathbf{1/2}$. Comparison between multiindices is performed componentwise; for example, $\mathbf{i} \geq \mathbf{j}$ means that $i_k \geq j_k$ for all k. The control points and B-spline outputs have subscripts that are multiindices. The continuous variables are written as vectors, $\mathbf{s} = (s_0, \ldots, s_{n-1})$ and $\mathbf{t} = (t_0, \ldots, t_{n-1})$.

Let the multidimensional image samples be $\{F_{\mathbf{i}}\}_{\mathbf{i}=\mathbf{0}}^{\mathbf{c}}$ and choose the degrees $\mathbf{d} > \mathbf{0}$. The B-spline interpolation is defined as a tensor produce spline,

$$\mathbf{X}_{\mathbf{d}+\mathbf{i}}(\mathbf{s}) = \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{d}} F_{\mathbf{i}+\mathbf{j}} \sum_{\mathbf{k}=\mathbf{0}}^{\mathbf{d}} \mathbf{a}_{\mathbf{jk}}^{(\mathbf{d})} (\mathbf{s} - \mathbf{d} - \mathbf{i})^{\mathbf{k}} \tag{45}$$

for $\mathbf{0} \leq \mathbf{i} \leq \mathbf{c}$ and $\mathbf{s} \in [\mathbf{d}+\mathbf{i}, \mathbf{d}+\mathbf{i}+\mathbf{1})$. The term $\mathbf{a}_{\mathbf{jk}}^{(\mathbf{d})}$ denotes the product of the n blending matrices $a_{j_k k_\ell}^{(d_\ell)}$ and the notation $(\mathbf{s} - \mathbf{d} - \mathbf{i})^{\mathbf{k}}$ denotes the product of the n terms $(s_\ell - d_\ell - i_\ell)^{k_\ell}$. The implementation will require you to choose $\mathbf{t} \in [-\mathbf{1/2}, \mathbf{c}+\mathbf{1/2}]$ and then compute internally $\mathbf{s}-\mathbf{d} = ((\mathbf{c}+\mathbf{1}-\mathbf{d})/(\mathbf{c}+\mathbf{1}))(\mathbf{t}+\mathbf{1/2})$. The indices $\mathbf{i}$ are determined from $\mathbf{s}$ in order to select $\mathbf{X}_{\mathbf{d}+\mathbf{i}}(\mathbf{s})$ for evaluation.

In the implementations, the summations of the B-spline evaluations typically involve nested loops, the depth of nesting depending on the dimension n. Although in a language such as C++ where you can generate code recursively using template metaprogramming, doing so with large n might exceed the compiler's capabilities. Instead, it is possible to implement the n-dimensional splines by careful memory organization and bookkeeping.

8.1 Multidimensionsal Array Layout

Consider an n-dimensional array of elements, each element located by an n-tuple $(x_0, x_1, \ldots, x_{n-1})$ where $0 \leq x_j < b_j$ for user-defined upper bounds b_j. The n-tuples can be mapped uniquely to 1-dimensional indices,

$$i = x_0 + b_0 x_1 + b_0 b_1 x_2 + \cdots + (b_0 \cdots b_{n-2}) x_{n-1}, \quad 0 \leq i < (b_0 \cdots b_{n-1}) \tag{46}$$

The coefficient of x_0 is 1 and the coefficient of x_j for $j > 0$ is $\prod_{k=0}^{j-1} b_k$. Given the 1-dimensional index i, the n-tuple can be extracted by a sequence of mod and div operations,

$$\mathbf{x} = (i \bmod b_0, (i \operatorname{div} b_0) \bmod b_1, ((i \operatorname{div} b_0) \operatorname{div} b_1) \bmod b_2, \ldots) \tag{47}$$

Listing 17 contains pseudocode for the conversions of equations (46) and (47).

Listing 17. Conversions from tuples to indices and from indices to tuples for an n-dimensional lattice. The upper bounds are `b[]` and the tuple to convert is `x[]`.

```
// Convert from a tuple to an index.  The conversion equation has a nested
// factoring similar to that for Horner's method for polynomial evaluation.
int GetIndex(int b[n], int x[n])
{
    int i = x[n-1];
    for (int k = n-2; k >= 0; --k)
    {
        i = b[k] * i + x[k];
    }
    return i;
}

// Convert from an index to a tuple.  The code is structured to use the
// minimum number of MOD and DIV operations.
void GetTuple(int b[n], int i, int x[n])
{
    for (int k = 0; k < n-1; ++k)
    {
        x[k] = i MOD b[k];
        i = i DIV b[k];
    }
    x[n-1] = i;
}
```

The conversion from an index to a tuple is relatively expensive because of the MOD and DIV operations. For a large number of B-spline evaluations, these operations will be a noticeable bottleneck reported by a profiler. The conversion is designed for random access. For code that iterates over the tuples in a structured manner, we can avoid the MOD and DIV operations as shown in the next section.

8.2 Eliminating Nested Loops

The ideas for replacing nested loops by a single loop are illustrated for dimensions 2 and 3.

Consider the double loop

```
for (int x1 = 0; x1 < b1; ++x1)
{
    for (int x0 = 0; x0 < b0; ++x0)
    {
        // work involving (x0,x1) goes here
    }
}
```

The implied ordering of the tuples is $(0,0)$, $(1,0)$, ..., $(b_0-1,0)$, $(0,1)$, $(1,1)$, ..., $(2b_0-1,0)$, ..., and so on. The 2-tuple pairs effectively traverse a 2D matrix in row-major order. The equivalent 1-dimensional index traversal is 0, 1, 2, ..., b_0b_1-1. The conversion to a single loop using MOD and DIV operations is

```
for (int i = 0; i < b0 * b1; ++i)
{
    int x0 = i MOD b0;
    int x1 = i DIV b1;

    // work involving (x0,x1) goes here
}
```

For the first b_0 values of i, x_0 is incremented from 0 to b_0-1 but x_1 is always 0. For the second b_0 values of i, x_0 again is incremented from 0 to b_0-1 but x_1 is always 1. A faster alternative is to increment x_0 from 0 to b_0-1 and, when it reaches b_0, wrap it around to 0 and increment x_1 as shown next

```
int x0 = 0, x1 = 0;
for (int i = 0; i < b0 * b1; ++i)
{
    // work involving (x0,x1) goes here

    // code that supports the elimination of nested loops
    if (++x0 < b0)
    {
        // x0 has been incremented to a value smaller than b0, so x1
        // remains its current value.
        continue;
    }

    // x0 has been incremented to b0, so wrap it to zero and
    // increment x1 to its next value.
    x0 = 0;
    ++x1;
}
```

Consider the triple loop

```
for (int x2 = 0; x2 < b2; ++x2)
{
    for (int x1 = 0; x1 < b1; ++x1)
    {
        for (int x0 = 0; x0 < b0; ++x0)
        {
            // work involving (x0,x1,x2) goes here
        }
    }
}
```

The implied ordering of the tuples is the usual lexicographical ordering that generalizes row-major order. The value x_0 varies the fastest, x_1 varies slower and x_2 varies the slowest. The conversion to a single loop using MOD and DIV operations is

```
for (int i = 0; i < b0 * b1 * b2; ++i)
{
    int x0 = i MOD b0;
    int temp = i / DIV b1;
    int x1 = i MOD b1;
    int x2 = i DIV b2;

    // work involving (x0,x1,x2) goes here
}
```

For the first $b_0 b_1$ iterations, x_2 is 0. Of those iterations, x_0 and x_1 vary in the same manner as illustrated previously for the double loop. The use of MOD and DIV adds computational overhead that is not necessary. A faster alternative is

```
int x0 = 0, x1 = 0, x2 = 0;
for (int i = 0; i < b0 * b1 * b2; ++i)
{
    // work involving (x0,x1,x2) goes here

    // code that supports the elimination of nested loops
    if (++x0 < b0)
    {
        // x0 has been incremented to a value smaller than b0, so x1
        // and x2 remain their current values.
        continue;
    }

    // x0 has been incremented to b0, so wrap it to zero and
    // increment x1 to its next value.
    x0 = 0;
    if (++x1 < b1)
```

```
    {
        // x1 has been incremented to a value smaller than b1, so x2
        // remains its current value.
        continue;
    }

    // x1 has been incremented to b1, so wrap it to zero and
    // increment x2 to its next value.
    x1 = 0;
    ++x2;
}
```

In the general case for n nested loops, the replacement by a single loop is shown in Listing 18.

Listing 18. Replacing n nested loops by a single loop. The upper bounds b[] are specified by the user. The number of n-tuples is quantity which is the product of all the bounds b[].

```
// n nested loops
for (x[n-1] = 0; x[n-1] < b[n-1]; ++x[n-1])
for (x[n-2] = 0; x[n-2] < b[n-2]; ++x[n-2])
 :
for (x[0] = 0; x[0] < b[0]; ++x[0])
{
    // work involving (x[0], x[1], ..., x[n-1]) goes here
}

// equivalent slow single loop
int x[n];
for (int i = 0; i < quantity; ++i)
{
    int temp = i;
    for (int d = 0; d < n-1; ++d)
    {
        x[d] = temp MOD b[d];
        temp = temp DIV b[d];
    }
    x[n-1] = temp;

    // work involving (x[0], x[1], ..., x[n-1]) goes here
}

// equivalent fast single loop, the x[] must be set to zero initially
for (int d = 0; d < n; ++d)
{
    x[d] = 0;
}
for (int i = 0; i < quantity; ++i)
{
    // work involving (x[0], x[1], ..., x[n-1]) goes here

    // code that supports the elimination of nested loops
    for (int d = 0; d < n; ++d)
    {
        if (++x[d] < b[d])
        {
            break;
        }
        x[d] = 0;
    }
}
```

Although not needed for the B-spline application, it is possible to generalize the conversion from nested loops to a single loop when there is also a lower bound; that is, when $\ell_j \le x_j < b_j$ for all j. Other variations of the pseudocode shown in Listing 18 are possible that manage nested oops with slightly more complicated logic. One such case is shown in the caching code for general dimensions.

8.3 Evaluation without Caching

The straightforward evaluation algorithm is to compute the polynomial terms first and then compute the weighted sum of control points. Equation (48) shows the form of the B-spline for this approach.

$$\mathbf{X_{d+i}(s)} = \sum_{\mathbf{j=0}}^{\mathbf{d}} F_{\mathbf{i+j}} \mathbf{\Phi_j(u)}, \quad \mathbf{\Phi_j(u)} = \sum_{\mathbf{k=0}}^{\mathbf{d}} \mathbf{a_{jk}^{(d)} u^k}, \quad \mathbf{u = s - d - i} \tag{48}$$

The B-spline derivatives of order $\mathbf{m} = (m_0, \ldots, m_{n-1})$ are

$$\mathbf{X_{d+i}^{(m)}(s)} = \sum_{\mathbf{j=0}}^{\mathbf{d}} F_{\mathbf{i+j}} \mathbf{\Phi_j^{(m)}(u)} \tag{49}$$

where $\mathbf{\Phi_j^{(m)}(u)}$ denotes the multiindexed sequence of derivatives of the polynomial components of $\mathbf{\Phi_j(u)}$.

Listings 9 and 13 show that only the evaluation loops themselves are nested. The other loops are single loops over the number of dimensions of the spline. Listing 19 shows the generic form of the evaluation code if one were to use nested loops.

Listing 19. Pseudocode for evaluation of equation (49) with nested loops.

```
//  degree[n]: degrees of the B-spline
//  numControls[n]: numbers of control point
//  controls: array of control points
//  A[n]: blending matrix of size (d[]+1)-by-(d[]+1), stored as A[](row,col)
//  dcoefficient[n]:  array of derivative polynomial coefficients, size (d[]+1)*(d[]+2)/2
//  lmax[n]:  location of the last coefficient for the block of specified order
//  phi[n]:  array of polynomial evaluations that multiply the control points
//  tmin[n]:  -1/2
//  tmax[n]:  (numControlPoints[] + 1/2 )
//  powerDSDT[n]:  array of powers of ds/dt = (numControls[] - d) / numControls[]

Controls::Type EvaluateNoCaching(int order[n], Real t[n])
{
    Controls::Type result = ctZero;
    int degreeMinusOrder[n];
    for (int dim = 0; dim < n; ++dim)
    {
        degreeMinusOrder[dim] = degree[dim] - order[dim];
        if (degreeMinusOrder[dim] < 0 || degreeMinusOrder[dim] > degree[dim])
        {
            return result;
        }
    }

    int i[n];
    Real u[n];
    for (int dim = 0; dim < n; ++dim)
    {
        GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
            i[dim], u[dim]);
    }

    for (int dim = 0; dim < n; ++dim)
    {
        for (int j = 0; j <= degree[dim]; ++j)
        {
            phi[dim][j] = 0;
            for (int k = degree[dim], ell = lmax[dim][order[dim]]; k >= order[dim]; --k, --ell)
            {
```

```
                phi[dim][j] = phi[dim][j] * u[dim] + A[dim](j, k) * dcoefficient[dim][ell];
            }
        }
    }

    // TODO: Replace the nested loops by a fast single loop.
    Real p[n], product[n];
    int j[n];
    for (j[n-1] = 0; j[n-1] <= degree[n-1]; ++j[n-1])
    {
        p[n-1] = phi[n-1][j[n-1]];
        product[n-1] = p[n-1];
        for (j[n-2] = 0; j[n-2] <= degree[n-2]; ++j[n-2])
        {
            p[n-2] = phi[n-2][j[n-2]];
            product[n-2] = p[n-2] * product[n-1];
            :
                for (j[0] = 0; j[0] <= degree[0]; ++j[0])
                {
                    p[0] = phi[0][j[0]];
                    product[0] = p[0] * product[1];
                    result = result + controls(i[0] + j[0], ..., i[n-1] + j[n-1]) * product[0];
                }
        }
    }

    Real adjust = 1;
    for (int dim = 0; dim < n; ++dim)
    {
        adjust *= powerDSDT[dim][order[dim]];
    }
    result = result * adjust;
    return result;
}
```

All the assignments of p[] and the multiplications for product[] can be moved inside the inner-most loop so that the loop marked TODO is of the form in Listing 18 for the fast single loop, after which we may replace the nested loops. However, a small modification of the fast loops avoids moving that code. The assigments of p[] occur only when the j-loop counters change.

It is also important to note that the controls provided by the application must have an accessor that allows you to pass an n-tuple in order to get the corresponding control point. Although the B-spline implementation can force controls to take a 1-dimensional index, this would impose a policy that the multidimensional control points be managed according to the mapping between n-tuples and 1-dimensional indices. The design of the GTE B-spline interpolation is such that a user provides a wrapper for the control points that hides the user's organization of control points. That wrapper must implement an accessor that takes an n-tuple and returns a control point; what the wrapper does with the tuple is not the interpolator's concern.

Listing 20 shows the new pseudocode without the nested loops and with the preparation of an n-tuple to pass to the controls object.

Listing 20. Pseudocode for evaluation of equation (49) without nested loops and with preparation of the n-tuple input to controls.

```
//  degree[n]: degrees of the B-spline
//  numControls[n]: numbers of control point
//  controls: array of control points
//  A[n]: blending matrix of size (d[]+1)-by-(d[]+1), stored as A[](row,col)
//  dcoefficient[n]:  array of derivative polynomial coefficients, size (d[]+1)*(d[]+2)/2
//  lmax[n]:  location of the last coefficient for the block of specified order
```

```
//  phi[n]:  array of polynomial evaluations that multiply the control points
//  tmin[n]:  -1/2
//  tmax[n]:  (numControlPoints[] + 1/2 )
//  powerDSDT[n]:  array of powers of ds/dt = (numControls[] - d) / numControls[]
//  kmax:  precomputed product of (degree[]+1) terms

Controls::Type EvaluateNoCaching(int order[n], Real t[n])
{
    Controls::Type result = ctZero;
    for (int dim = 0; dim < n; ++dim)
    {
        if (order[dim] < 0 || order[dim] > degree[dim])
        {
            return result;
        }
    }

    int i[n];
    Real u[n];
    for (int dim = 0; dim < n; ++dim)
    {
        GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
            i[dim], u[dim]);
    }

    for (int dim = 0; dim < n; ++dim)
    {
        for (int j = 0; j <= degree[dim]; ++j)
        {
            phi[dim][j] = 0;
            for (int k = degree[dim], ell = lmax[dim][order[dim]]; k >= order[dim]; --k, --ell)
            {
                phi[dim][j] = phi[dim][j] * u[dim] + A[dim](j, k) * dcoefficient[dim][ell];
            }
        }
    }

    int j[n], sumIJ[n];
    Real p[n];
    for (int dim = 0; dim < n; ++dim)
    {
        j[dim] = 0;
        sumIJ[dim] = i[dim];
        p[dim] = phi[dim][0];
    }
    for (int k = 0; k < kmax; ++k)
    {
        Real product = 1;
        for (int dim = 0; dim < n; ++dim)
        {
            product *= p[dim];
        }

        result = result + controls(sumIJ) * product;

        for (int dim = 0; dim < n; ++dim)
        {
            if (++j[dim] <= degree[dim])
            {
                p[dim] = phituple[dim][j[dim]];
                sumIJ[dim] = ituple[dim] + j[dim];
                break;
            }
            j[dim] = 0;
            p[dim] = phituple[dim][0];
            sumIJ[dim] = ituple[dim];
        }
    }

    Real adjust = 1;
    for (int dim = 0; dim < n; ++dim)
    {
```

```
        adjust *= powerDSDT[dim][order[dim]];
    }
    result = result * adjust;
    return result;
}
```

8.4 Evaluation with Caching

Equation (45) is rewritten for this algorithm,

$$\mathbf{X_{d+i}(s)} = \sum_{\mathbf{k=0}}^{\mathbf{d}} \mathbf{\Phi_{ik} u^k}, \quad \mathbf{\Phi_{ik}} = \sum_{\mathbf{j=0}}^{\mathbf{d}} \mathbf{F_{i+j} a_{jk}^{(d)}} \tag{50}$$

The tensor $\mathbf{\Phi_{ik}}$ has $2n$ indices for dimension n, n for multindex $\mathbf{i}$ and n for multiindex $\mathbf{k}$. Listing 21 contains pseudocode for precomputing the tensor.

Listing 21. Pseudocode for precomputing $\mathbf{\Phi_{ik}}$. The numLocalControls is the product of degree[]+1 terms, which is the size of the neighborhood of control points used for an evaluation. The algorithm for eliminating nested loops is used in this code.

```
void ComputeTensor(int i[n], int k[n], int index)
{
    Controls::Type element = ctZero;
    int j[n], sumIJ[n];
    for (int dim = 0; dim < n; ++dim)
    {
        j[dim] = 0;
    }
    for (int iterate = 0; iterate < numLocalControls; ++iterate)
    {
        Real blend = 1;
        for (int dim = 0; dim < n; ++dim)
        {
            blend *= A[dim](j[dim], k[dim]);
            sumIJ[dim] = i[dim] + j[dim];
        }
        element = element + controls(sumIJ) * blend;

        for (int dim = 0; dim < n; ++dim)
        {
            if (++j[dim] <= degree[dim] + 1)
            {
                break;
            }
            j[dim] = 0;
        }
    }
    tensor[index] = element;
}

void InitializeTensors()
{
    int tbound[2 * n];  // first n for i-tuple, second n for k-tuple
    int current = 0;
    int numCached = 1;
    for (int dim = 0; dim < n; ++dim, ++current)
    {
        tbound[current] = degree[dim] + 1;
        numCached *= tbound[current];
    }
```

```
    for (int dim = 0; dim < n; ++dim, ++current)
    {
        tbound[current] = numControls[dim] - degree[dim];
        numCached *= tbound[current];
    }
    Controls::Type tensor(numCached);

    int tuple[2 * n];
    for (int dim = 0; dim < n; ++dim)
    {
        tuple[dim] = 0;
    }
    for (int index = 0; index < numCached; ++index)
    {
        ComputeTensor(&tuple[n], &tuple[0], index);
        for (int i = 0; i < 2 * n; ++i)
        {
            if (++tuple[i] < tbound[i])
            {
                break;
            }
            tuple[i] = 0;
        }
    }
}
```

Listing 22 contains pseudocode for the evaluation of equation (50).

Listing 22. Pseudocode for evaluation the n-dimensional B-spline function using precaching.

```
Controls::Type EvaluateCaching(int const* order, Real const* t)
{
    // The numIterates is the number of tensor elements to combine
    // in a neighborhood with dimensions (degree[]-order[]+1).
    int numIterates = 1;
    for (int dim = 0; dim < n; ++dim)
    {
        if (order[dim] < 0 || order[dim] > degree[dim])
        {
            return ctZero;
        }
        numIterates *= degree[dim] - order[dim] + 1;
    }

    int i[n];
    Real u[n];
    for (int dim = 0; dim < n; ++dim)
    {
        GetKey(t[dim], tmin[dim], tmax[dim], powerDSDT[dim][1], numControls[dim], degree[dim],
            i[dim], u[dim]);
    }

    // Get the portion of the 1-dimensional index into the tensor corresponding
    // to the i multiindex.
    int iIndex = i[n - 1];
    int j1 = 2 * n - 2;
    for (int j0 = n - 2; j0 >= 0; --j0, --j1)
    {
        iIndex = tbound[j1] * iIndex + i[j0];
    }
    iIndex = tbound[j1] * iIndex;

    int j[n], k[n], ell[n];
    Controls::Type term[n];
    for (int dim = 0; dim < n; ++dim)
    {
        j[dim] = 0;
```

```
        k[dim] = degree[dim];
        ell[dim] = LMax[dim][order[dim]];
        term[dim] = ctZero;
    }

    for (int iterate = 0; iterate < numIterates; ++iterate)
    {
        // Get the portion of the 1-dimensional index into the tensor corresponding
        // to the k multiindex and combine it with the one from the i multiindex
        // to obtain the full index.
        int index = iIndex + k[n - 1];
        for (int j0 = n - 2; j0 >= 0; --j0)
        {
            index = tbound[j0] * index + k[j0];
        }

        if (cacheMode == ON_DEMAND_CACHING && !cached[index])
        {
            ComputeTensor(i, k, index);
            cached[index] = true;
        }
        term[0] = term[0] * u[0] + tensor[index] * dcoefficient[0][ell[0]];

        for (int dim = 0; dim < n; ++dim)
        {
            if (++j[dim] <= degree[dim] - order[dim])
            {
                --k[dim];
                --ell[dim];
                break;
            }
            int dimp1 = dim + 1;
            if (dimp1 < n)
            {
                term[dimp1] = term[dimp1] * u[dimp1] + term[dim] * mdcoefficient[dimp1][ell[dimp1]];
                term[dim] = ctZero;
                j[dim] = 0;
                k[dim] = degree[dim];
                ell[dim] = LMax[dim][order[dim]];
            }
        }
    }
    Controls::Type result = term[n - 1];

    Real adjust = 1;
    for (int dim = 0; dim < n; ++dim)
    {
        adjust *= powerDSDT[dim][order[dim]];
    }
    result = result * adjust;
    return result;
}
```

References

[1] Elaine Cohen, Richard F. Riesenfeld, and Gershon Elber. *Geometric Modeling with Splines*. AK Peters, Ltd, Natick, MA, 2001.

[2] Wolfram Research, Inc. *Mathematica 12.1.1.0*. Wolfram Research, Inc., Champaign, Illinois, 2020.