B-Spline Interpolation on Lattices

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: July 15, 1999
Last Modified: March 2, 2008

Contents

1 Introduction

It is sufficient to understand spline interpolation on 1-dimensional tabular data with uniformly spaced input values. Spline interpolation of higher dimensional tabular data on a uniform grid becomes a tensor product of 1-dimensional interpolating functions.

Given a table of function values $\{f_i\}_{i=0}^{N-1}$ where the inputs are the uniformly spaced integers $0 \leq i \leq N-1$, we want to build a spline of degree D which smoothly approximates the function values. The spline should have local control, not because we want to move the sample points around, but because an exact matching spline requires solving large sparse matric systems to determine the spline coefficients. One must also decide if the spline should have the convex hull property (like B-splines, but not Catmull-Rom interpolation).

Let $B(x)$ be the spline function of a single variable x. It is defined piecewise on intervals of the form $[i, i+1)$ for $1 \leq i \leq N-D$. The restriction on i avoids having to introduce boundary conditions for the interpolation, which would require special handling. For $x \in [i, i+1)$, the spline is defined as

$$B(x) = f_{i-1+j} m_{jk} X_k(x)$$

where the summation ranges are $0 \leq j \leq D$ and $0 \leq k \leq D$. The polynomial vector is

$$X(x) = \left(1, (x-i), (x-i)^2, \ldots, (x-i)^D\right).$$

The $(D+1) \times (D+1)$ *blending matrix* $M = m_{jk}$ can be chosen for any desired interpolation scheme. For B-splines I will provide in the next subsection an algorithm where the input is D and the output is M. The evaluation of the B-spline for a given x involves nested summation over the appropriate indices. Derivatives of $B(x)$ are evaluated accordingly:

$$\frac{d^m B(x)}{dx^m} = f_{i-1+j} m_{jk} \frac{d^m X_k(x)}{dx^m}$$

where $1 \leq m \leq D$. For $m > D$, the derivatives are identically zero since $B(x)$ is a piecewise polynomial of degree D.

Spline interpolation for 2-dimensional tables $\{f_{ij}\}_{i=0,j=0}^{N-1,M-1}$ are built as a tensor product. Let $B(x, y)$ be the spline function. It is defined piecewise on squares of the form $[i, i+1) \times [j, j+1)$ for $1 \leq i \leq N-D$ and $1 \leq j \leq M-D$. For $(x, y) \in [i, i+1) \times [j, j+1)$, the spline is defined by

$$B(x, y) = f_{i-1+k, j-1+\ell} m_{kr} m_{\ell s} X_r(x) Y_s(y)$$

where

$$\begin{aligned} X(x) &= \left(1, (x-i), (x-i)^2, \ldots, (x-i)^D\right) \\ Y(y) &= \left(1, (y-j), (y-j)^2, \ldots, (y-j)^D\right) \end{aligned}$$

Evaluation of $B(x, y)$ and its partial derivatives are straightforward as in the 1-dimensional case.

For higher dimensions, the construction is easily extended. For example, in 3 dimensions the spline function for $(x, y, z) \in [i_0, i_0+1) \times [i_1, i_1+1) \times [i_2, i_2+1)$ is

$$B(x, y, z) = f_{i_0-1+j_0, i_1-1+j_1, i_2-1+j_2} m_{j_0 k_0} m_{j_1 k_1} m_{j_2 k_2} X_{k_0}(x) Y_{k_1}(y) Z_{k_2}(z)$$

where

$$\begin{aligned} X(x) &= \left(1, (x-i_0), (x-i_0)^2, \ldots, (x-i_0)^D\right) \\ Y(y) &= \left(1, (y-i_1), (y-i_1)^2, \ldots, (y-i_1)^D\right) \\ Z(z) &= \left(1, (z-i_2), (z-i_2)^2, \ldots, (z-i_2)^D\right) \end{aligned}$$

2 B-Spline Blending Matrices

Here is a short discussion of recursive generation of the basis functions for B-splines. The main result is the construction of the blending matrix when the input values of the tabular data are uniformly spaced integers.

Given values $\{t_i\}_{i=0}^n$, define

$$B_i^0(t) = \left\{ \begin{array}{ll} 1, & t \in [t_i, t_{i+1}] \\ 0, & \text{otherwise} \end{array} \right\}.$$

Recursively define

$$B_i^d(t) = \left(\frac{t - t_i}{t_{i+d} - t_i} \right) B_i^{d-1}(t) + \left(\frac{t_{i+d+1} - t}{t_{i+d+1} - t_{i+1}} \right) B_{i+1}^{d-1}(t)$$

for $d \geq 1$. For $t_i = i$ (uniformly spaced integers), $B_{i+j}^d(t) = B_i^d(t - j)$, so there is essentially one B-spline basis function to compute for each d, call it $B_d(t)$. Thus,

$$B_0(t) = \left\{ \begin{array}{ll} 1, & t \in [0, 1] \\ 0, & \text{otherwise} \end{array} \right\},$$

and

$$B_{d+1}(t) = \frac{t}{d+1} B_d(t) + \frac{d+2-t}{d+1} B_d(t-1)$$

where

$$B_d(t) = \left\{ \begin{array}{ll} P_d^{(k)}(t), & t \in [k, k+1) \text{ for } 0 \leq k \leq d \\ 0 & \text{otherwise} \end{array} \right\}$$

where the $P_d^{(k)}(t)$ are polynomials of degree d (to be determined). The recursion now implies

$$P_0^{(0)}(t) = 1$$

and

$$P_{d+1}^{(k)}(t) = \frac{t}{d+1} P_d^{(k)}(t) + \frac{d+2-t}{d+1} P_d^{(k-1)}(t-1), \quad 0 \leq k \leq d+1.$$

Now let

$$P_d^{(k)}(t) = \frac{1}{d!} \sum_{i=0}^{d} a_i^{(k,d)} t^i.$$

Then

$$P_d^{(k)}(t-1) = \frac{1}{d!} \sum_{i=0}^{d} a_i^{(k,d)} (t-1)^i = \frac{1}{d!} \sum_{i=0}^{d} \left(\sum_{j=i}^{d} (-1)^{j-i} \left(\begin{array}{c} j \\ i \end{array} \right) a_j^{(k,d)} \right) t^i =: \frac{1}{d!} \sum_{i=0}^{d} b_i^{(k,d)} t^i.$$

The recursion for the polynomials now yields

$$a_i^{(k,d+1)} = a_{i-1}^{(k,d)} - b_{i-1}^{(k-1,d)} + (d+2) b_i^{(k-1,d)}$$

for $0 \leq i \leq d+1$ and $0 \leq k \leq d$. By convention, if an index is out of range, the term containing that index is 0. The initial data is $a_0^{(0,0)} = 1$ (so $b_0^{(0,0)} = 1$).

Now define

$$Q_d^{(k)}(t) = P_d^{(k)}(t+k);$$

then

$$Q_d^{(k)}(t) = \frac{1}{d!}\sum_{i=0}^{d} a_i^{(k,d)}(t+k)^i = \frac{1}{d!}\sum_{i=0}^{d}\left(\sum_{j=i}^{d} k^{j-i}\begin{pmatrix} j \\ i \end{pmatrix} a_j^{(k,d)}\right) t^i =: \frac{1}{d!}\sum_{i=0}^{d} c_i^{(k,d)} t^i.$$

The blending matrix $M = m_{ij}$ used in the B-spline tensor (of degree d) calculation is

$$m_{ij} = c_i^{(d-j,d)}.$$

Blending matrices for some small d values are

$$d = 1: \quad \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

$$d = 2: \quad \tfrac{1}{2}\begin{bmatrix} 1 & -2 & 1 \\ 1 & 2 & -2 \\ 0 & 0 & 1 \end{bmatrix}$$

$$d = 3: \quad \tfrac{1}{6}\begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$d = 4: \quad \tfrac{1}{24}\begin{bmatrix} 1 & -4 & 6 & -4 & 1 \\ 11 & -12 & -6 & 12 & -4 \\ 11 & 12 & -6 & -12 & 6 \\ 1 & 4 & 6 & 4 & -4 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$d = 5: \quad \tfrac{1}{120}\begin{bmatrix} 1 & -5 & 10 & -10 & 5 & 1 \\ 26 & -50 & 20 & 20 & -20 & 5 \\ 66 & 0 & -60 & 0 & 30 & -10 \\ 26 & 50 & 20 & -20 & -20 & 10 \\ 1 & 5 & 10 & 10 & 5 & -5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The routine `BlendMatrix` in the base class `Spline` (in file `spline.c`) implements the recursions described above.

3 Direct Implementation

The direct implementations of B-splines in 1D and 2D are given here. For higher dimensions, the tensor summations require more nested `for` loops. For an n-dimensional problem the calculations require $2n$ nested loops. In the next subsection I'll go through a series of steps to optimize the code and remove the nested loops. The result is reduced address calculation which yields highly optimized code in time. The cost is increased memory usage which is used to store data in an appropriate order to gain the time decrease.

3.1 1D Splines

The spline function $B(x) = f_{\lfloor x \rfloor - 1 + j} m_{jk} X_k(x)$ and its derivative $B'(x)$ are calculated as follows.

```
const int degree;           // degree > 0
const int dp1 = degree+1;
const int dim;              // dim >= degree+1
float m[dp1][dp1];          // B-spline matrix
float data[dim];            // 1D data set
float x;                    // 1 <= x < dim-degree+1

// determine interval [bx,bx+1) for evaluation
int bx = floor(x);

// compute intermediate tensor (non-polynomial part of B(x))
float inter[dp1];
for (int kx = 0; kx <= degree; kx++) {
    inter[kx] = 0;
    for (int ix = bx-1, jx = 0; jx <= degree; ix++, jx++)
        inter[kx] += data[ix]*m[jx][kx];
}

// polynomial storage for spline and its derivatives
float px[dp1];

//----- compute B(x) -----

// compute polynomial part (1,x,x^2,x^3,...,x^{degree})
px[0] = 1;
for (kx = 1; kx <= degree; kx++)
    px[kx] = px[kx-1]*(x-bx);

// compute final result
float result = 0;
for (kx = 0; kx <= degree; kx++)
    result += inter[kx]*px[kx];

//----- compute B'(x) -----
```

```
// compute polynomial part (0,1,2x,3x^2,...,degree*x^{degree-1})
px[0] = 0;
px[1] = 1;
for (kx = 2; kx <= degree; kx++)
    poly[kx] = kx*poly[kx-1]*(x-bx);

// compute final result
float result = 0;
for (kx = 0; kx <= degree; kx++)
    result += inter[kx]*px[kx];

//----- other derivatives can be similarly computed -----
```

3.2 2D Splines

The spline function $B(x,y) = f_{\lfloor x \rfloor -1+k, \lfloor y \rfloor -1+\ell} m_{kr} m_{\ell s} X_r(x) Y_s(y)$ and its derivative $B_x(x,y)$ are calculated as follows.

```
const int degree;            // degree > 0
const int dp1 = degree+1;
const int xdim;              // xdim >= degree+1
const int ydim;              // ydim >= degree+1
float m[dp1][dp1];           // B-spline matrix
float data[ydim][xdim];      // 2D data set
float x;                     // 1 <= x < xdim-degree+1
float y;                     // 1 <= y < ydim-degree+1

// determine interval [base0,base0+1)x[base1,base1+1) for evaluation
int bx = floor(x);
int by = floor(y);

// compute intermediate tensor (non-polynomial part of B(x,y))
float inter[dp1][dp1];
for (int ky = 0; ky <= degree; ky++)
for (int kx = 0; kx <= degree; kx++) {
    inter[ky][kx] = 0;
    for (int iy = by-1, jy = 0; jy <= degree; iy++, jy++)
    for (int ix = bx-1, jx = 0; jx <= degree; ix++, jx++)
        inter[ky][kx] += data[iy][ix]*m[jx][kx]*m[jy][ky];
}

// polynomial storage for spline and its derivatives
float px[dp1];
float py[dp1];

//----- compute B(x,y) -----
```

```
// compute x polynomial part (1,x,x^2,x^3,...,x^{degree})
px[0] = 1;
for (kx = 1; kx <= degree; kx++)
    px[kx] = px[kx-1]*(x-bx);

// compute y polynomial part (1,y,y^2,y^3,...,y^{degree})
py[0] = 1;
for (ky = 1; ky <= degree; ky++)
    py[ky] = py[ky-1]*(y-by);

// compute final result
float result = 0;
for (ky = 0; ky <= degree; ky++)
for (kx = 0; kx <= degree; kx++)
    result += inter[ky][kx]*px[kx]*py[ky];

//----- compute B_x(x,y) -----

// compute x polynomial part (1,x,x^2,x^3,...,x^{degree})
px[0] = 1;
for (kx = 1; kx <= degree; kx++)
    px[kx] = px[kx-1]*(x-bx);

// compute y polynomial part (0,1,2x,3x^2,...,degree*x^{degree-1})
py[0] = 0;
py[1] = 1;
for (ky = 2; ky <= degree; ky++)
    py[ky] = ky*py[ky-1]*(y-by);

// compute final result
float result = 0;
for (ky = 0; ky <= degree; ky++)
for (kx = 0; kx <= degree; kx++)
    result += inter[ky][kx]*px[kx]*py[ky];

//----- other derivatives can be similarly computed -----
```

4 Generalized and Optimized Spline Calculation

The first of my goals is to create a spline class for which the user can specify the dimension of the data set dynamically. In the 2D example, the dimension is used statically in three ways. The first way is the structure of the data set and the intermediate tensor as `float[][]`. The second way is the occurrence of nested loops, two at a time. The third way is that the innermost loop of the intermediate calculation requires multiplication by n matrix values.

The second of my goals is to optimize the spline calculations for speed. The bottleneck in the spline evaluation is the calculation of the intermediate tensor, which in general involves $2n$ nested loops.

The next subsections present modifications to meet the three goals. As each modification is made, the revised code for the 2-dimensional case is given.

4.1 Multidimensional Array Layout

An n-dimensional array can be stored in memory as a 1-dimensional array of contiguous bytes. The elements of the array are ordered lexicographicaly. Let the data set be defined on an n-dimensional lattice $L = \prod_{d-0}^{n-1}[0, b_d)$ where $[0, b_d) = \{i \in \mathbb{Z} : 0 \leq i < b_d\}$ for $b_d > 0$. That is, the data set is a function $f : L \to \mathbb{R}$. If $f(\mathbf{x})$ is the data value at coordinate $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$, then the corresponding 1-dimensional index is

$$i = x_0 + b_0 x_1 + b_0 b_1 x_2 + \cdots (b_0 \cdots b_{n-2}) x_{n-1}.$$

The coefficient of x_k is $\prod_{d=0}^{k-1} b_k$ (if $k = 0$, the product defaults to 1). Given a 1-dimensional index i, the corresponding coordinate can be obtained by a sequence of mod and div operations:

$$\mathbf{x} = (i \bmod b_0, (i \text{ div } b_0) \bmod b_1, ((i \text{ div } b_0) \text{ div } b_1) \bmod b_2, \ldots).$$

The data and intermediate tensor can be stored as 1-dimensional arrays. The relevant changes in the 2D code are

```
float data[xdim*ydim];    // f(i,j) = data[i+xdim*j]

// compute intermediate tensor
float inter[dp1*dp1];
for (int ky = 0; ky <= degree; ky++)
for (int kx = 0; kx <= degree; kx++) {
    float sum = 0;
    for (int iy = by-1, jy = 0; jy <= degree; iy++, jy++)
    for (int ix = bx-1, jx = 0; jx <= degree; ix++, jx++)
        sum += data[ix+xdim*iy]*m[jx][kx]*m[jy][ky];
    inter[kx+dp1*ky] = sum;
}

// compute B(x,y)
float result = 0;
for (ky = 0; ky <= degree; ky++)
for (kx = 0; kx <= degree; kx++)
    result += inter[kx+dp1*ky]*px[kx]*py[ky];
```

A small modification to the loops can avoid the address calculation `kx+dp1*ky`. This works because of the lexicographical ordering of `inter` and the ordering of the “k” loops.

```
// compute intermediate tensor
int iindex = 0;
for (int ky = 0; ky <= degree; ky++)
for (int kx = 0; kx <= degree; kx++) {
    float sum = 0;
    for (int iy = by-1, jy = 0; jy <= degree; iy++, jy++)
    for (int ix = bx-1, jx = 0; jx <= degree; ix++, jx++)
        sum += data[ix+xdim*iy]*m[jx][kx]*m[jy][ky];
    inter[iindex++] = sum;
}

// compute B(x,y)
float result = 0;
iindex = 0;
for (ky = 0; ky <= degree; ky++)
for (kx = 0; kx <= degree; kx++)
    result += inter[iindex]*px[kx]*py[ky];
```

A similar modification to avoid the address calculation `ix+xdim*iy` is not straightforward since the data accessed is a subrectangle of the original data set whose columns (x) are stored in contiguous memory for each row, but whose rows (y) are not contiguous. It is, however, not necessary to do so. The method used to avoid a static number of matrix values `m[j][k]` in the innermost loop will require the data to be *cached* into a contiguous block of memory. Anticipating this, the intermediate tensor code becomes

```
// cache the data
float cache[dp1*dp1];
int cindex = 0;
for (int iy = by-1; iy <= by-1+degree; iy++)
for (int ix = bx-1; ix <= bx-1+degree; ix++)
    cache[cindex] = data[ix+xdim*iy];

// compute intermediate tensor
int iindex = 0;
for (int ky = 0; ky <= degree; ky++)
for (int kx = 0; kx <= degree; kx++) {
    float sum = 0;
    cindex = 0;
    for (int jy = 0; jy <= degree; jy++)
    for (int jx = 0; jx <= degree; jx++)
        sum += cache[cindex]*m[jx][kx]*m[jy][ky];
    inter[iindex++] = sum;
}
```

4.2 Condensing Matrix Products

The next step is to remove the explicit matrix product `m[jx][kx] * m[jy][ky]` which forces a static dependence on dimension 2. Let i_0 and i_1 be given (which is the case for a single spline evaluation). The intermediate second-order tensor is

$$I_{k_0 k_1} = m_{j_0 k_0} m_{j_1 k_1} f_{i_0 - 1 + j_0 \; i_1 - 1 + j_1}.$$

In the code we have the correspondences

symbols	variables
i_0	`bx`
i_1	`by`
$i_0 - 1 + j_0$	`ix`
$i_1 - 1 + j_1$	`iy`
j_0	`jx`
j_1	`jy`
k_0	`kx`
k_1	`ky`

The matrix product can be viewed abstractly as a fourth-order tensor

$$P_{j_0 j_1 k_0 k_1} = m_{j_0 k_0} m_{j_1 k_1}.$$

For given i_0 and i_1 (which is the case for a single spline evaluation), the cached data is a second-order tensor

$$C_{j_0 j_1} = f_{i_0 - 1 + j_0 \; i_1 - 1 + j_1}.$$

The intermediate tensor is therefore

$$I_{k_0 k_1} = P_{j_0 j_1 k_0 k_1} C_{j_0 j_1}.$$

Let J be the 1-dimensional index corresponding to the coordinate (j_0, j_1). If D is the degree of the spline, then $0 \le J < D^2$ and J is mapped to the coordinate via $J = j_0 + D * j_1$. Similarly K corresponds to (k_0, k_1) with $0 \le K < D^2$ and $K = k_0 + D * k_1$. The intermediate tensor can be written in multiindex notation as

$$I_K = P_{JK} C_J$$

where the right-hand side is a matrix P times a vector C. The intermediate code, which additionally stores P as a 1D array, now becomes

```
// cache the data
float cache[dp1*dp1];
int cindex = 0;
for (int iy = by-1; iy <= by-1+degree; iy++)
for (int ix = bx-1; ix <= bx-1+degree; ix++)
    cache[cindex] = data[ix+xdim*iy];

// compute product tensor (order of loops is important)
float product[dp1*dp1*dp1*dp1];
int pindex = 0;
for (int k1 = 0; k1 <= degree; k1++)
for (int k0 = 0; k0 <= degree; k0++)
for (int j1 = 0; j1 <= degree; j1++)
```

```
    for (int j0 = 0; j0 <= degree; j0++)
        product[pindex] = m[j0][k0]*m[j1][k1];

    // compute intermediate tensor
    const int dp1_to_2 = dp1*dp1;
    for (int K = 0; K < dp1_to_2; K++) {
        float sum = 0;
        for (int J = 0; J < dp1_to_2; J++)
            sum += product[J+dp1_to_2*K]*cache[J];
        inter[K] = sum;
    }
```

A minor modification can be made to remove the calculation of `J+dp1_to_2*K`.

```
    pindex = 0;
    for (int K = 0; K < dp1_to_2; K++) {
        float sum = 0;
        for (int J = 0; J < dp1_to_2; J++, pindex++)
            sum += product[pindex]*cache[J];
        inter[K] = sum;
    }
```

Note that (in general) the original $2n$ intermediate tensor loops are reduced to 2 loops, thereby reducing address calculations. The cost is the increased space required to store the product tensor.

After these changes, the intermediate tensor calculation is still the bottleneck in the calculations. The calculation time can be further reduced. The $(D+1) \times (D+1)$ B-spline matrix has at least D zeros (the ones occurring in the last row). The presence of these zeros forces many of the product tensor entries to be zero. For example,

dimension	degree	zeros	total	ratio
2	1	7	16	0.438
	2	32	81	0.395
	3	112	256	0.438
	4	184	625	0.294
	5	455	1296	0.351
3	1	37	64	0.578
	2	386	729	0.529
	3	2368	4096	0.578
	4	6364	15625	0.407
	5	22267	46656	0.477
4	1	175	256	0.684
	2	4160	6561	0.634
	3	44800	65536	0.684
	4	196144	390625	0.502

A significant portion of time is spent updating the running sum in the innermost loop with zero. We can add another fourth-order tensor (of integers) to the code. This tensor measures how many of the product tensor entries to skip to get to the next nonzero value. That is, if P_i is the product tensor stored as a 1-dimensional array, then define the skip tensor by $S_i = j$ if $P_i \neq 0$, $P_{i+j} \neq 0$, and $P_k = 0$ for $i < k < i + j$. Note that if $P_i \neq 0$ and $P_{i+1} \neq 0$, then $S_i = 1$. If $P_i = 0$, the value of S_i is irrelevant, but I will set it to 1. The code to calculate the skips is

```
// compute product tensor (order of loops is important)
int dp1_to_4 = dp1*dp1*dp1*dp1;
float product[dp1_to_4];
int pindex = 0;
for (int k1 = 0; k1 <= degree; k1++)
for (int k0 = 0; k0 <= degree; k0++)
for (int j1 = 0; j1 <= degree; j1++)
for (int j0 = 0; j0 <= degree; j0++)
    product[pindex] = m[j0][k0]*m[j1][k1];

// compute skip tensor
float skip[dp1_to_4];
for (int i = 0; i < dp1_to_4; ) {
    skip[i] = 1;
    for (int j = 0; j < dp1_to_4 && product[j] == 0; j++)
        skip[j]++;
    i = j;
}
```

```
// compute intermediate tensor (order of increment of J and I important)
for (int K = 0, I = 0; K < dp1_to_2; K++) {
    float sum = 0;
    for (int J = 0; J < dp1_to_2; J += skip[I], I += skip[I])
        sum += product[I]*cache[J];
    inter[K] = sum;
}
```

The product and skip tensors can be computed at time of spline object creation.

4.3 Eliminating Nested Loops

The general idea of converting N nested loops to a single loop is described. Consider the example

```
int L0, L1;  // specified lower bounds
int U0, U1;  // specified upper bounds, L0 < U0 and L1 < U1

// 2 nested loops
for (int i1 = L1; i1 < U1; i1++)
for (int i0 = L0; i0 < U0; i0++) {
    // loop body goes here
}
```

One way to convert to a single loop is to use the conversions between 1-dimensional indices and 2-dimensional coordinates. Using the lexicographic ordering $j = (i_0 - L_0) + b_0 * (i_1 - L_1)$ where $b_0 = U_0 - L_0$,

```
// equivalent single loop
int b0 = U0-L0, b1 = U1-L1, quantity = b0*b1;
for (int j = 0; j < quantity; j++) {
    i0 = L0 + (j % b0);
    i1 = L1 + (j / b0);

    // loop body goes here
}
```

The general case of N loops is

```
const int N;      // dimensions, N > 0
int L[N], U[N];  // bounds, L[d] < U[d]
int i[N];         // loop indices, L[d] <= i[d] < U[d]

// N nested loops
for (i[N-1] = L[N-1]; i[N-1] < U[N-1]; i[N-1]++)
for (i[N-2] = L[N-1]; i[N-2] < U[N-2]; i[N-2]++)
 :
```

```
for (i[0] = L[0]; i[0] < U[0]; i[0]++) {
    // loop body goes here
}

// equivalent single loop
int b[N];
int quantity = 1;
for (int d = 0; d < N; d++) {
    b[d] = U[d]-L[d];
    quantity *= b[d];
}

for (int j = 0; j < quantity; j++) {
    int temp = j;
    for (d = 0; d < N; d++) {
        i[d] = L[d] + temp % b[d];
        temp /= b[d];
    }

    // loop body goes here
}
```

In exchange for a single loop, the execution time increases because of coordinate calcuations.

Another reduction scheme is described now. The previous one made conversions from indices to coordinates which work for a *random* index. The algorithm does not take into account that the indices are processed in the order corresponding to the lexicographical ordering of the coordinates. Consider the same example as before,

```
int L0, L1;  // specified lower bounds
int U0, U1;  // specified upper bounds, L0 < U0 and L1 < U1

// 2 nested loops
for (int i1 = L1; i1 < U1; i1++)
for (int i0 = L0; i0 < U0; i0++) {
    // loop body goes here
}

// equivalent single loop
int b0 = U0-L0, b1 = U1-L1, quantity = b0*b1;
i0 = L0;
i1 = L1;
for (int j = 0; j < quantity; j++) {
    // loop body goes here

    if ( ++i0 < U0 )
        continue;
    i0 = 0;
```

```
        i1++;  // j==quantity when i1==U1, so only j-test needed
    }
```

In the general case we have the reduction

```
    const int N;     // dimensions, N > 0
    int L[N], U[N];  // bounds, L[d] < U[d]
    int i[N];        // loop indices, L[d] <= i[d] < U[d]

    // N nested loops
    for (i[N-1] = L[N-1]; i[N-1] < U[N-1]; i[N-1]++)
    for (i[N-2] = L[N-2]; i[N-2] < U[N-2]; i[N-2]++)
     :
    for (i[0] = L[0]; i[0] < U[0]; i[0]++) {
        // loop body goes here
    }

    // equivalent single loop
    int quantity = 1;
    for (int d = 0; d < N; d++)
        quantity *= U[d]-L[d];

    for (d = 0; d < N; d++)
        i[d] = L[d];

    for (int j = 0; j < quantity; j++) {
        // loop body goes here

        for (d = 0; d < N; d++) {
            if ( ++i[d] < U[d] )
                break;
            i[d] = L[d];
        }
    }
```

4.4 Reduction of Product Tensor Loops

The first reduction method is applied to computing the product tensor from the spline matrix. For dimension N, $2N$ nested loops are required

```
    const int N;  // dimensions
    int dp1_to_2N = 1;
    for (int d = 0; d < N; d++)
        dp1_to_2N *= dp1;
    dp1_to_2N *= dp1_to_2N;

    // compute product tensor (order of loops is important)
```

```
float product[dp1_to_2N];
int pindex = 0;
int j[N], k[N];
for (k[N-1] = 0; k[N-1] <= degree; k[N-1]++)
:
for (k[0] = 0; k[0] <= degree; k[0]++)
for (j[N-1] = 0; j[N-1] <= degree; j[N-1]++)
:
for (j[0] = 0; j[0] <= degree; j[0]++)
    product[pindex] = m[j[0]][k[0]]* ... * m[j[N-1]][k[N-1]];
```

These can be reduced to

```
int i[2*N];
for (int j = 0; j < dp1_to_2N; j++) {
    int temp = j;
    for (d = 0; d < 2*N; d++) {
        i[d] = temp % dp1;
        temp /= dp1;
    }
    product[j] = 1;
    for (int k = 0; k < N; k++)
        product[k] *= m[i[k]][i[k+N]];
}
```

Because of the nested loop ordering, the $2N$-dimensional coordinate `i[]` contains j_0 through j_{N-1} in its first N components and k_0 through k_{N-1} in its last N components.

4.5 Reduction of Cache Tensor Loops

This reduction uses the second method, but it needs a minor modification to extract the sublattice of data points. The 2-dimensional code is

```
int ix = bx-1, iy = by-1, dindex = ix+xdim*iy;
for (int cindex = 0; cindex < dp1_to_2; cindex++, dindex++) {
    cache[cindex] = data[dindex];

    if ( ++ix <= bx-1+degree )
        continue;
    ix = bx-1;
    dindex += xdim-dp1;

    iy++;
}
```

The data sublattice is a rectangle of size $(D+1) \times (D+1)$. When the index `ix` points to the last column in a row, the next incrementing of it takes it out-of-range, so it is reset to the first column in the next row `ix`

= bx-1. At the same time the data index dindex was incremented to an out-of-range column (immediately to the right of the data set). The addition of xdim to dindex gets you to the same out-of-range column, but in the next row. The subtraction of dp1 from dindex gets you to the first column in that next row.

The extension of this idea to N-dimensions is

```
const int N;       // dimensions, N > 0
int degree;        // degree D of spline, D > 0
int dp1;           // D+1
int dp1_to_N;      // (D+1)^N
int dim[N];        // dimension sizes, dim[d] > 0
int grid_min[N];  // coordinate of first data point to be processed
int grid_max[N];  // grid_max[d] = grid_min[d]+degree
int i[N];

for (int d = 0; d < N; d++)
    i[d] = grid_min[d];

// compute 1-dim starting index from N-dim starting coordinate
int dindex = i[N-1];
for (d = N-2; d >= 0; d--) {
    dindex *= dim[d];
    dindex += i[d];
}

// compute data address differences, used when loop var out-of-range
int delta[N-1];
int temp = 1;
for (d = 0; d < N-1; d++) {
    delta[d] = temp*(dim[d]-dp1);
    temp *= dim[d];
}

for (int cindex = 0; cindex < dp1_to_N; cindex++, dindex++) {
    cache[cindex] = data[dindex];

    for (d = 0; d < N; d++) {
        if ( i[d] <= grid_max[d] )
            break;
        i[d] = grid_min[d];
        dindex += delta[d];
    }
}
```

4.6 Reduction of Evaluation Loops

Consider the example of evaluating a B-spline in two variables. The final loops in the process are shown below.

```
// intermediate tensor computations go here
// polynomial computations for px[] and py[] go here

// compute B(x,y)
float result = 0;
int iindex = 0;
for (int ky = 0; ky <= degree; ky++)
for (int kx = 0; kx <= degree; kx++)
    result += inter[iindex]*px[kx]*py[ky];
```

The loops can be replaced using the second reduction method:

```
int i0 = 0, i1 = 0;
float result = 0;
for (int j = 0; j < dp1_to_2; j++) {
    result += inter[j]*px[i0]*py[i1];

    if ( i0 <= degree )
        continue;
    i0 = 0;
    i1++;
}
```

In N dimensions, the reduction is

```
float poly[N][dp1];  // N variables, each poly of degree D+1

int i[N];
float result = 0;
for (int j = 0; j < dp1_to_N; j++) {
    float temp = inter[j];
    for (int d = 0; d < N; d++)
        temp *= poly[d][i[d]];
    result += temp;

    for (d = 0; d < N; d++) {
        if ( i[d] <= degree )
            break;
        i[d] = 0;
    }
}
```

Now consider evaluating a derivative, say $B_x(x, y)$. We evaluated the appropriate polynomial terms, but used the same final set of loops. Note, however, that for a derivative in x of order m, the first m components of the polynomial array are zero. Thus, the loops could be written as shown to avoid the multiplications by zero.

```
// compute "dx" partial derivatives of B(x,y) with respect to x
// and "dy" partial derivatives of B(x,y) with respect to y

float result = 0;
for (ky = dy; ky <= degree; ky++)
for (kx = dx; kx <= degree; kx++)
    result += inter[ky][kx]*px[kx]*py[ky];
```

In the previous example, the loop index `j` is used to traverse all of the intermediate tensor values. To avoid the multiplications by zero, now `j` needs to skip over blocks of intermediate tensor values which would have been the other operands of the zero multiplication. The code is

```
int dx, dy;  // orders of differentation
    // px[0] = ... = px[dx-1] = 0
    // py[0] = ... = px[dy-1] = 0
int i0 = dx, i1 = dy;
int delta0 = dx;  // amount of "skip" in the inter[] tensor

float result = 0;
for (int j = i0+dp1*i1; j < dg1_to_N; j+) {
    result += inter[j]*px[i0]*py[i1];

    if ( ++i0 <= degree )
        continue;
    i0 = dx;
    j += delta0;  // skip the first dx inter[] values in the next row
    i1++;
}
```

In N dimensions, the reduction is

```
float poly[N][dp1];  // N variables, each poly of degree D+1

int dx[N];  // orders of differentiation for vars x0 through x_{N-1}
    // poly[d][0] = ... = poly[d][dx[d]-1] = 0

// loop indices
int i[N];
for (int d = 0; d < N; d++)
    i[d] = dx[d];

// skip amounts for inter[] tensor
```

```
int delta[N-1];
int temp = 1;
for (d = 0; d < N-1; d++) {
    delta[d] = temp*dx[d];
    temp *= dp1;
}

// starting index
int j = i[N-1];
for (d = N-2; d >= 0; d--) {
    j *= dp1;
    j += i[d];
}

float result = 0;
for ( ; j < dp1_to_N; j++) {
    float temp = inter[j];
    for (int d = 0; d < N; d++)
        temp *= poly[d][i[d]];
    result += temp;

    for (d = 0; d < N; d++) {
        if ( i[d] <= degree )
            break;
        i[d] = dx[d];
        j += delta[d];
    }
}
```

5 Polynomial Construction

It is sufficient to consider the 1-dimensional case since each polynomial is independent of the others. The following table illustrates what the polynomial arrays will store. The value $t = x - \lfloor x \rfloor$ and D is the degree of the polynomial.

derivative order	polynomial array							
0	(	1,	t,	t^2,	t^3,	$\ldots$,	t^D	)
1	(	0,	1,	$2t$,	$3t^2$,	$\ldots$,	Dt^{D-1}	)
2	(	0,	0,	2,	$6t$,	$\ldots$,	$D(D-1)t^{D-2}$	)
$\vdots$	$\vdots$							
D	(	0,	0,	0,	0,	$\ldots$,	$D!$	)
$D+1$	(	0,	0,	0,	0,	$\ldots$,	0	)

To allow for a dynamically selected degree D, we need to construct the coefficients for the polynomials above. We can do so by building a $(D+1)\times(D+1)$ matrix, $[\alpha_{rc}]$ which stores the coefficients indicated in the table above. The matrix is upper triangular with row index $0 \le r \le D+1$ indicating order of differentiation and column index $0 \le c \le D+1$ indicating the power of t that goes with the coefficient. It is given by

$$\alpha_{rc} = \left\{ \begin{array}{ll} 1 & c \ge r = 0 \\ \prod_{i=0}^{r-1}(c-i) & c \ge r > 0 \end{array} \right\}.$$

The code implementing this is

```
float coeff[dp1][dp1];
for (int r = 0; r <= degree; r++)
    for (int c = r; c <= degree; c++) {
        coeff[r][c] = 1;
        for (i = 0; i < r; i++)
            coeff[r][c] *= c-i;
    }
```

The code to compute the polynomial array is

```
float poly[dp1];
float t = x-floor(x);
int order = dx;  // order of differentation

float temp = 1;
for (int d = order; order <= degree; d++) {
    poly[d] = coeff[order][d]*temp;
    temp *= t;
}
```

6 Avoiding Intermediate Calculations

The order of calculations in the evaluation of B-splines was: compute floors of the input variables to use as base indices, compute the intermediate tensor product, compute the polynomial terms, multiply the intermediate tensor product with the polynomials and return the result. Even with the optimizations discussed earlier, the intermediate tensor calculation is still the bottleneck. For random selection of input variables, this tensor must always be calculated. In many applications, the evaluation of splines is localized. Given that the spline is evaluated at $\mathbf{x}$, the probability is large that the next evaluation is at $\mathbf{y}$ where $|\mathbf{x}-\mathbf{y}|$ is smaller than one unit distance relative to the lattice on which the data is defined. Using the locality of reference, it is highly probable that the base indices are the same for both $\mathbf{x}$ and $\mathbf{y}$. If we update the intermediate tensor only when a base index changes, we could save some significant time. The code to do this is given. Variable names are consistent with what was used earlier in this section.

```
//----- initialization code -----
```

```
const int N;      // dimensions, N > 0
int old_base[N];  // saves the base indices from the previous evaluation
int base[N];      // current base indices, 1 <= base[d] < N-degree+1
for (int d = 0; d < N; d++)
    old_base[d] = -1;  // base[d] will be different on first call

//----- code per evaluation -----

float x[N];  // input to spline evaluation

for (d = 0; d < N; d++)
    base[d] = floor(x[d]);  // compute base indices for current call
for (d = 0; d < N; d++)
    if ( old_base[d] != base[d] ) {  // base index has changed
        // switch to new local grid
        for (int k = 0; k < N; k++) {
            old_base[k] = base[k];
            grid_min[k] = base[k]-1;
            grid_max[k] = grid_min[k]+degree;
        }

        // compute intermediate tensor code goes here
        break;
    }

// compute polynomial terms code goes here

// final calculations go here (multiplication of polynomials and
// intermediate tensor)
```

7 Computing Data On-Demand

For some applications, the tabular data might not be completely known when the B-spline object is to be constructed. The reason most likely is that the precalculation of the entire data set is very expensive. In these applications it is economical to compute only those local grid points needed for evaluation. For a N-dimensional data set, each dimension requiring K values, the total number of grid points is K^N. Precalculation takes order $O(K^N)$ time. If the application only needs to analyze data lying approximately on a curve in the data set, the time to process this localized data is order $O(DK)$ where D is the degree of the spline. Clearly this is a large time savings.

The user of a spline object can provide a callback function which is executed whenever tabular data have not yet been calculated. The callback has the job of computing the tabular data. A mechanism to handle this is as follows. Any unknown tabular data are assigned an invalid floating point number. I use `MAXFLOAT` for this purpose. If for some reason the application allows `MAXFLOAT` as valid data, then some invalid bit pattern for IEEE short reals should be chosen. Whenever a new local grid is entered, the spline object checks for invalid data and executes the callback for that data. For a 2-dimensional spline class `Spline2`,

```
// "data" is the tabular data pointer used by "spline"
// "data" is of size xdim-by-ydim

void Spline2::EvaluateUnknownData ()
{
    for (int i1 = grid_min[1]; i1 <= grid_max[1]; i1++)
    for (int i0 = grid_min[0]; i0 <= grid_max[0]; i0++) {
        int index = i0+xmin*i1;
        if ( data[index] == INVALID_DATA )
            data[index] = evaluate_callback(i0,i1);
    }
}
```

Th callback function pointer is assigned to spline object member function `evaluate_callback)` by the user. The method for general dimensions is

```
int SplineN::Index (int* i)
{
    int index = i[N-1];
    for (int d = N-2; d >= 0; d--) {
        index *= dp1;
        index += i[d];
    }
    return index;
}

void SplineN::EvaluateUnknownData ()
{
    int i[N];
    for (int d = 0; d < N; d++)
        i[d] = grid_min[d];

    for (int j = 0; j < dp1_to_N; j++) {
        int index = Index(i);
        if ( data[index] == INVALID_DATA )
            data[index] = evaluate_callback(index);

        for (d = 0; d < N; d++) {
            if ( ++i[d] <= grid_max[d] )
                break;
            i[d] = grid_min[d];
        }
    }
}
```

Only the code block for computing the intermediate tensor product needs the tabular data. The evaluation code (in general dimensions) now looks like

```
// Spline provides member function pointer,
//     float (*evaluate_callback)(int*)
// for storing the callback.  Pointer is null unless set by user.

float x[N];  // input to spline evaluation

for (d = 0; d < N; d++)
    base[d] = floor(x[d]);
for (d = 0; d < N; d++)
    if ( old_base[d] != base[d] ) {
        // switch to new local grid
        for (int k = 0; k < N; k++) {
            old_base[k] = base[k];
            grid_min[k] = base[k]-1;
            grid_max[k] = grid_min[k]+degree;
        }

        // compute unknown data
        if ( evaluate_callback )
            EvaluateUnknownData();

        // compute intermediate tensor code goes here
        break;
    }

// compute polynomial terms code goes here

// final calculations go here (multiplication of polynomials and
// intermediate tensor)
```

In general, `EvaluateUnknownData()` checks the entire $(D+1)^N$ set of grid points for invalid data. When the base indices are incremented or decremented by at most 1, it is not necessary to check all values. If you want to optimize this routine, it would go as follows. Suppose that any of the pairs of old and new base indicies differ by at most 1. For simplicity, assume that new data members are added to the proper classes which keep track of these differences, The function computations can be modified to

```
// 1D splines
int bx_diff = bx-oldbx;

void EvaluateUnknownData ()
{
    int k0 = bx+2*bx-_diff+1;  // k0 = bx-1 or bx+3
    if ( data[k0] == INVALID_FLOAT )
        data[k0] = evaluate_callback(k0);
}

// 2D splines
int bx_diff = bx-oldbx;
int by_diff = by-oldby;

void EvaluateUnknownData ()
{
    int k0, k1;

    if ( bx_diff ) {
        k0 = bx+2*bx_diff+1;  // k0 = bx-1 or bx+3
        for (k1 = grid_min[1]; k1 <= grid_max[1]; k1++) {
            int index = k0+xmin*k1;
            if ( data[index] == INVALID_DATA )
                data[index] = evaluate_callback(k0,k1);
        }
    }

    if ( by_diff ) {
        k1 = by+2*by_diff+1;  // k1 = by-1 or by+3
        for (k0 = grid_min[0]; k0 <= grid_max[0]; k0++) {
            int index = k0+xmin*k1;
            if ( data[index] == INVALID_DATA )
                data[index] = evaluate_callback(k0,k1);
        }
    }
}
```

Note that there are still some duplicate checks, but the total number of checks is reduced from $(D+1)^n$ to at most $n*(D+1)^{n-1}$. Rather than checking the entire "volume" of grid points, we are only checking the new "faces" that occur when the base indices change. The additional check occurs at the intersection of the faces. I have not implemented this modification since in low dimensions it probably does not gain any time.

8 Putting It All Together

I have constructed the spline classes by derivation from an abstract base spline class. The base class has the job of initializing things such as the spline blending matrix, the polynomial coefficient matrix, the product and skip tensors for optimization, and the base indices. The class declaraion is in `spline.h`:

```
class mgcSpline
{
public:
    mgcSpline (int _dimensions, int _degree, int* _dim, float* _data);
    virtual ~mgcSpline ();

    int Dimensions () { return dimensions; }
    int Degree () { return degree; }
    const float** Matrix () { return (const float**)matrix; }
    float DomainMin (int d = 0) { return dom_min[d]; }
    float DomainMax (int d = 0) { return dom_max[d]; }
    int GridMin (int d = 0) { return grid_min[d]; }
    int GridMax (int d = 0) { return grid_max[d]; }
    float (*evaluate_callback)(int);

    static const float INVALID_DATA;

    // spline evaluation for function interpolation (no derivatives)
    virtual float operator() (float* x) = 0;

    // spline evaluation, derivative counts given in dx[]
    virtual float operator() (int* dx, float* x) = 0;

protected:
    int dimensions; // N, number of dimensions
    int degree;     // D, degree of polynomial spline
    int Dp1;        // D+1
    int Dp1_to_N;   // power(D+1,N)
    int Dp1_to_2N;  // power(D+1,2N)
    int* dim;       // dimension sizes dim[0] through dim[N-1]
    float* data;    // N-dimensional array of data
    float* dom_min; // minimum allowed value of spline input vector
    float* dom_max; // maximum allowed value of spline input vector
    int* grid_min;  // minimum allowed value for current local grid
    int* grid_max;  // maximum allowed value for current local grid
    int* base;      // base indices for grid of local control points
    int* old_base;  // old base indices for grid of local control points
    float* cache;   // cache for subblock of data
    float* inter;   // intermediate product of data with blending matrix
    float** poly;   // poly[N][D+1] for storing polynomials and derivatives
    float** coeff;  // coefficients for polynomial construction
    float** matrix; // (D+1)x(D+1) blending matrix
    float* product; // outer tensor product of matrix with itself N times
    int* skip;      // for skipping zero values of mtensor

    virtual void EvaluateUnknownData () = 0;
    virtual void ComputeIntermediate () = 0;
    void SetPolynomial (int order, float diff, float* poly);

private:
    void Create ();
    static int Choose (int n, int k);
    static float** BlendMatrix (int deg);

// error handling
public:
    static int verbose;
    static unsigned error;
```

```
        static void Report (ostream& ostr);
    private:
        static const unsigned invalid_dimensions;
        static const unsigned invalid_degree;
        static const unsigned invalid_dim;
        static const unsigned null_data;
        static const unsigned allocation_failed;
        static const char* message[];
        static int Number (unsigned single_error);
        static void Report (unsigned single_error);
    };
```

The static member INVALID_DATA is initialized to MAXFLOAT in the spline source file. Method SetPolynomial evaluates polynomial terms. Method Create does the allocation and initialization of the various arrays. Method Choose computes the combinatorial value of "n choose k". Method BlendMatrix computes the blending matrix described in the theory of B-splines. The member function pointer evaluate_callback is initialized to null by the constructor. If your application requires data to be computed on-demand, then set this pointer to the function that generates your data. If necessary, you'll need to map the input index to an n-dimensional coordinate if your function is defined on coordinates. Eventually I should put in an additional pointer to a callback that takes as input a coordinate. The evaluation operators then need to check if either pointer is not null. The method EvaluateUnknownData, which calls the callback, also needs to be duplicated so that one handles the index-callback, the other handles the coordinate-callback.

8.1 1D Splines

The class declaration for 1D splines is in spline1.h:

```
    class mgcSpline1 : public mgcSpline
    {
    public:
        mgcSpline1 (int _degree, int _dim, float* _data);

        // spline evaluation for function interpolation (no derivatives)
        float operator() (float x);
        float operator() (float* x) { return (*this)(*x); }

        // spline evaluation, derivative count given in dx
        float operator() (int dx, float x);
        float operator() (int* dx, float* x) { return (*this)(*dx,*x); }

    private:
        static const int N;

        void EvaluateUnknownData ();
        void ComputeIntermediate ();
    };
```

The static variable N conveniently stores the constant dimension 2. An example of how to use the class to build a bicubic spline curve is given below.

```
int main ()
{
    int degree = 3;
    int dim = 100;  // number of curve points to interpolate
    float xdata[100], ydata[100];  // the points, set them to whatever

    mgcSpline1 x(degree,dim,xdata), y(degree,dim,ydata);

    // calculate a point on the bicubic spline curve
    cout << "point is (" << x(50) << ’,’ << y(50) << ’)’ <, endl;

    // calculate a tangent at the point
    cout << "tangent is (" << x(1,50) << ’,’ << y(1,50) << ’)’ <, endl;

    return 0;
}
```

8.2 2D Splines

The class declaration for 2D splines is in `spline2.h`:

```
class mgcSpline2 : public mgcSpline
{
public:
    mgcSpline2 (int _degree, int* _dim, float* _data);

    int Index (int x, int y) { return x+dim[0]*y; }

    // spline evaluation for function interpolation (no derivatives)
    float operator() (float* x);
    float operator() (float x, float y);

    // spline evaluation, derivative counts given in dx[]
    float operator() (int* dx, float* x);
    float operator() (int dx, int dy, float x, float y);

private:
    static const int N;

    void EvaluateUnknownData ();
    void ComputeIntermediate ();
};
```

The static variable N conveniently stores the constant dimension 2. The method Index converts 2D coordinates to 1D indices. An example of using the class is shown below. This one uses the callback mechanism to compute data on-demand.

```
    int quantity;
    float Callback (int index)
    {
        return exp(index/float(quantity));
    }

    int main ()
    {
        int degree = 3;
        int dim[2] = { 8, 8 };
        quantity = dim[0]*dim[1];

        int x[2];
        float* data = new float[quantity];
        for (x[1] = 0; x[1] < dim[1]; x[1]++)
        for (x[0] = 0; x[0] < dim[0]; x[0]++) {
            int index = x[0]+dim[0]*x[1];
            data[index] = mgcSpline::INVALID_DATA;
        }

mgcSpline2 f(degree,dim,data);
        f.evaluate_callback = Callback;

        float rx[2];
        for (x[1] = f.DomainMin(1); x[1] <= f.DomainMax(1); x[1]++) {
            rx[1] = x[1];
            for (x[0] = f.DomainMin(0); x[0] <= f.DomainMax(0); x[0]++) {
                rx[0] = x[0];
                cout << f(rx) << ' ';
            }
            cout << endl;
        }

        delete[] data;
        return 0;
    }
```

8.3 3D Splines

The class declaration for 3D splines is in spline3.h:

```
    class mgcSpline3 : public mgcSpline
    {
```

```
public:
    mgcSpline3 (int _degree, int* _dim, float* _data);

    int Index (int x, int y, int z)
        { return x+dim[0]*(y+dim[1]*z); }

    // spline evaluation for function interpolation (no derivatives)
    float operator() (float* x);
    float operator() (float x, float y, float z);

    // spline evaluation, derivative counts given in dx[]
    float operator() (int* dx, float* x);
    float operator() (int dx, int dy, int dz, float x, float y, float z);

private:
    static const int N;

    void EvaluateUnknownData ();
    void ComputeIntermediate ();
};
```

The static variable N conveniently stores the constant dimension 3. The method Index converts 3D coordinates to 1D indices. An example of using the class is shown below. This one precomputes the data.

```
int main ()
{
    int dim[3] = { 8, 8, 8 };
    int quantity = dim[0]*dim[1]*dim[2];
    float* data = new float[quantity];
    mgcSpline3 f(3,dim,data);

    int x[3];
    for (x[2] = 0; x[2] < dim[2]; x[2]++)
    for (x[1] = 0; x[1] < dim[1]; x[1]++)
    for (x[0] = 0; x[0] < dim[0]; x[0]++) {
        int index = f.Index(x[0],x[1],x[2]);
        data[index] = exp(index/float(quantity));
    }

    int order[3] = { 1, 0, 2 };  // compute f_{xzz}(x,y,z)
    float rx[3];
    for (x[2] = f.DomainMin(2); x[2] <= f.DomainMax(2); x[2]++) {
        rx[2] = x[2];
        for (x[1] = f.DomainMin(1); x[1] <= f.DomainMax(1); x[1]++) {
            rx[1] = x[1];
            for (x[0] = f.DomainMin(0); x[0] <= f.DomainMax(0); x[0]++) {
                rx[0] = x[0];
                cout << f(order,rx) << ' ';
            }
            cout << endl;
        }
        cout << endl;
    }

    delete[] data;
    return 0;
}
```

8.4 4D Splines

The class declaration for 3D splines is in `spline3.h`:

```
class mgcSpline4 : public mgcSpline
{
public:
    mgcSpline4 (int _degree, int* _dim, float* _data);

    int Index (int x, int y, int z, int w)
        { return x+dim[0]*(y+dim[1]*(z+dim[2]*w)); }

    // spline evaluation for function interpolation (no derivatives)
    float operator() (float* x);
    float operator() (float x, float y, float z, float w);

    // spline evaluation, derivative counts given in dx[]
    float operator() (int* dx, float* x);
    float operator() (int dx, int dy, int dz, int dw, float x, float y,
    float z, float w);

private:
    static const int N;

    void EvaluateUnknownData ();
    void ComputeIntermediate ();
};
```

The static variable `N` conveniently stores the constant dimension 3. The method `Index` converts 4D coordinates to 1D indices. I'm sure you get the idea on the examples.

8.5 N-dimensional Splines

The general spline class which allows dynamic specification of both dimension N and degree D is declared in `splinen.h`:

```
class mgcSplineN : public mgcSpline
{
public:
    mgcSplineN (int _dimensions, int _degree, int* _dim, float* _data);
    ~mgcSplineN ();

    int Index (int* i);

    // spline evaluation for function interpolation (no derivatives)
    float operator() (float* x);

    // spline evaluation, derivative counts given in dx[]
    float operator() (int* dx, float* x);

private:
    const int N;
    int* ev_i;
    int* ci_loop;
    int* ci_delta;
    int* op_i;
    int* op_j;
    int* op_delta;

    void EvaluateUnknownData ();
    void ComputeIntermediate ();

// error handling
public:
    static int verbose;
    static unsigned error;
    static void Report (ostream& ostr);
private:
    static const unsigned allocation_failed;
    static const char* message[];
    static int Number (unsigned single_error);
    static void Report (unsigned single_error);
};
```

The variable `N` stores the number of dimensions which is constant for a given object. The method `Index` converts N-dimensional coordinates to a 1-dimensional index. The private pointers are used as abstract loop indices. Rather than allocate them every time a method is called, they are allocated once in the constructor. This class is much slower than the specialized ones because of the overhead of processing abstract loop indices.

9 Timing

I timed the intermediate tensor product calculations both for the direct implementation and the optimized one. The only difference between a direct class and the optimized one is the body of method `void ComputeIntermediate()`. The bodies in 2D are

```
//----- Optimized -----------------------------------------------------

// cache the data
static int delta0 = dim[0]-Dp1;
int loop[N];
for (int d = 0; d < N; d++)
    loop[d] = grid_min[d];
int index = Index(loop[0],loop[1]);
for (int k = 0; k < Dp1_to_N; k++, index++) {
    cache[k] = data[index];

    if ( ++loop[0] <= grid_max[0] )
        continue;
    loop[0] = grid_min[0];
    index += delta0;

    loop[1]++;
}

// compute and save the intermediate product
for (int i = 0, j = 0; i < Dp1_to_N; i++) {
    float sum = 0;
    for (k = 0; k < Dp1_to_N; k += skip[j], j += skip[j])
        sum += product[j]*cache[k];
    inter[i] = sum;
}


//----- Direct --------------------------------------------------------

int iindex = 0;
for (int ky = 0; ky <= degree; ky++)
for (int kx = 0; kx <= degree; kx++) {
    float sum = 0;
    for (int iy = base[1]-1, jy = 0; jy <= degree; iy++, jy++)
    for (int ix = base[0]-1, jx = 0; jx <= degree; ix++, jx++)
        sum += data[ix+dim[0]*iy]*matrix[jx][kx]*matrix[jy][ky];
    inter[iindex++] = sum;
}
```

The bodies in 3D are

```
//----- Optimized ------------------------------------------------------

// cache the data
static int delta[N-1] = { dim[0]-Dp1, dim[0]*(dim[1]-Dp1) };
int loop[N];
for (int d = 0; d < N; d++)
    loop[d] = grid_min[d];
int idx = Index(loop[0],loop[1],loop[2]);
for (int k = 0; k < Dp1_to_N; k++, idx++) {
    cache[k] = data[idx];

    if ( ++loop[0] <= grid_max[0] )
        continue;
    loop[0] = grid_min[0];
    idx += delta[0];

    if ( ++loop[1] <= grid_max[1] )
        continue;
    loop[1] = grid_min[1];
    idx += delta[1];

    loop[2]++;
}

// compute and save the intermediate product
for (int i = 0, j = 0; i < Dp1_to_N; i++) {
    float sum = 0;
    for (k = 0; k < Dp1_to_N; k += skip[j], j += skip[j])
        sum += product[j]*cache[k];
    inter[i] = sum;
}


//----- Direct ---------------------------------------------------------
int iindex = 0;
for (int kz = 0; kz <= degree; kz++)
for (int ky = 0; ky <= degree; ky++)
for (int kx = 0; kx <= degree; kx++) {
    float sum = 0;
    for (int iz = base[2]-1, jz = 0; jz <= degree; iz++, jz++)
    for (int iy = base[1]-1, jy = 0; jy <= degree; iy++, jy++)
    for (int ix = base[0]-1, jx = 0; jx <= degree; ix++, jx++)
        sum += data[ix+dim[0]*(iy+dim[1]*iz)]*
               matrix[jx][kx]*matrix[jy][ky]*matrix[jz][kz];
    inter[iindex++] = sum;
}
```

The bodies in 4D are

```
//----- Optimized ----------------------------------------------------

// fetch subblock of data to cache
static int delta[N-1] = {
    dim[0]-Dp1, dim[0]*(dim[1]-Dp1), dim[0]*dim[1]*(dim[2]-Dp1)
};
int loop[N];
for (int d = 0; d < N; d++)
    loop[d] = grid_min[d];
int index = Index(loop[0],loop[1],loop[2],loop[3]);
for (int k = 0; k < Dp1_to_N; k++, index++) {
    cache[k] = data[index];

    if ( ++loop[0] <= grid_max[0] )
        continue;
    loop[0] = grid_min[0];
    index += delta[0];

    if ( ++loop[1] <= grid_max[1] )
        continue;
    loop[1] = grid_min[1];
    index += delta[1];

    if ( ++loop[2] <= grid_max[2] )
        continue;
    loop[2] = grid_min[2];
    index += delta[2];

    loop[3]++;
}

// compute and save the intermediate product
for (int i = 0, j = 0; i < Dp1_to_N; i++) {
    float sum = 0;
    for (k = 0; k < Dp1_to_N; k += skip[j], j += skip[j])
        sum += product[j]*cache[k];
    inter[i] = sum;
}


//----- Direct -------------------------------------------------------

int iindex = 0;
for (int kw = 0; kw <= degree; kw++)
for (int kz = 0; kz <= degree; kz++)
for (int ky = 0; ky <= degree; ky++)
for (int kx = 0; kx <= degree; kx++) {
    float sum = 0;
```

```
        for (int iw = base[3]-1, jw = 0; jw <= degree; iw++, jw++)
        for (int iz = base[2]-1, jz = 0; jz <= degree; iz++, jz++)
        for (int iy = base[1]-1, jy = 0; jy <= degree; iy++, jy++)
        for (int ix = base[0]-1, jx = 0; jx <= degree; ix++, jx++)
            sum += data[ix+dim[0]*(iy+dim[1]*(iz+dim[2]*iw))]*
                   matrix[jx][kx]*matrix[jy][ky]*
                   matrix[jz][kz]*matrix[jw][kw];
        inter[iindex++] = sum;
    }
```

The experiment consisted of iterating `ComputeIntermediate` a large number of times and measuring the execution time. The table below summarizes the results. The times are in seconds. The experiment was performed on a Pentium 90 Mhz with 16 Megabytes of memory.

Dimension	Method	Iterations	Time	Speedup
2	direct	1,000,000	122.098	2.85
	optimized	1,000,000	42.787	
3	direct	10,000	27.682	3.97
	optimized	10,000	6.975	
4	direct	1,000	56.902	5.42
	optimized	1,000	10.491	